ALGORITHMIQUE II

El Maati CHABBAR

Département d'Informatique

Faculté des Science –Rabat-

PLAN DU COURS

OBJECTIF: **Conception des algorithmes corrects et efficaces**

PLAN

- RAPPELS : NOTATIONS ALGORITHMIQUES
- COMPLEXITE
- ALGORITHMES ITERATIFS DE TRIS
- RECURSIVITE
- DIVISER POUR RESOUDRE
- PREUVE D'ALGORITHMES

- Un type est un ensemble de valeurs sur lesquelles on définit des opérations.
  - Types de base :
    - ✓ **Entier** : Opérateurs arithmétiques +, -, *, div, mod
    - ✓ **Réel** : Opérateurs arithmétiques +, -, *, /
    - ✓ **Booléen** : Opérateurs logiques et, ou, non
    - ✓ **Caractère** : constante (lettre imprimable) entre apostrophe.
  - Les opérateurs relationnels permettant de faire des comparaisons: $<, \leq, =, >, \geq, \neq$

  Le Résultat de la comparaison est une valeur booléenne.

- Une **variable** possède :
  - un nom
  - une valeur
  - un type

(la valeur d'une variable peut changer au cours de l'exécution)

Déclaration : <variable> : <type>

- Une **expression**, pour un type, est soit une constante, soit une variable, soit constituée à l'aide de constantes, de variables, de parenthèses et des opérateurs

- INSTRUCTIONS
- **Affectation :** <variable> := <expression>

- **Condition :** **si** <condition> **alors**

  action

  fsi

ou :

- **si** <condition> **alors**

  action1

  sinon

  action 2

  fsi

(condition est une expression à valeur booléenne;

action est une instruction ou un bloc d'instructions séparées par ;)

- **Itération**
  - boucle Pour

pour <variable> := <initiale> **à** <final> **faire**
action
fpour

(Où initiale et finale sont des expressions de même type que celui de la variable contrôlant la boucle, le type peut être entier, caractère ou énuméré)

Remarque: la boucle **pour** affecte la valeur de initiale à variable et compare cette valeur à celle de finale avant d'exécuter action.

- Exemple: calcul de 1+2+…+n (n entier≥ 1 fixé)

```
Programme somme_des_n_premiersTermes
// partie déclaration
n : entier; s : entier; i : entier;  (ou n, i, s : entier)
Début
// Lecture des données
    Écrire (" n = ? " );  lire(n);
// calcul de la somme
    s := 0;
    pour i := 1 à n faire
            s := s + i;
    fpour;
// affichage du résultat
    écrire("1+ 2 + … + n =  ", s);
fin
```

- en C

```
main ( ) {
// déclaration des variables
    int i, n, s;
// lecture des données
    prinf("   n = ?  "); scanf(" %d" , &n);
// calcul de la somme
    s = 0;
    for(i = 1; i <=n; i++)  s = s + i;
// affichage
    printf(" 1+2+…+ %d = %d \n" , n, s)
}
```

- **Itération**

• <u>boucle tantque</u>

```
tantque  < condition>  faire
                action
 ftantque
```

• <u>boucle répéter</u>

```
répéter
    action
jusque  < condition>
```

Remarques:

- Condition est une expression à valeur booléenne, cette expression doit être modifiée dans le corps de la boucle (dans action).
- La boucle pour peut être traduite en boucle tantque (ou en répéter) mais l'inverse n'est pas toujours vrai.

```
Programme somme_des_n_premiersTermes
// partie déclaration
n : entier; s : entier; i : entier;  (ou n, i, s : entier)
Début
// Lecture des données
      Écrire (" n = ? " );  lire(n);
// calcul de la somme
      s := 0;
      i := 1;
      tantque i <= n faire
                  s := s + i;
                  i := i + 1;
      ftantque;
// affichage du résultat
      écrire("1+ 2 + … + n =  ", s);
fin
```

- en C

```
main ( ) {
// déclaration des variables
      int i, n, s;
// lecture des données
      prinf("   n = ?  "); scanf(" %d" , &n);
// calcul de la somme
      s = 0;
      i = 1;
      while (i <= n) {
                  s = s + i;
                  i = i + 1;
      }
// affichage
      printf(" 1+2+…+ %d = %d \n" , n, s)
}
```

Algorithme

- Description formelle d'un procédé de calcul qui permet, à partir d'un ensemble de données, d'obtenir des résultats.
- Succession finie d'opérations

DONNEES algorithme → **RESULTATS**

Cheminement à suivre :



Conception structurée des algorithmes

Découper l'algorithme en sous algorithmes plus simples. Chaque sous algorithme est un algorithme.

Fonctions

- Une fonction est un sous algorithme (sous programme) qui, à partir de données, retourne un seul type de résultat.
- Une fonction
  - possède un nom
  - communique avec l'extérieur par le biais des paramètres
  - retourne un résultat par l'instruction **retourner**(expression)

Schéma d'une fonction :

fonction <nom de la fonction>(Liste des paramètres) : <type du résultat>

// déclaration des variables locales

début

// corps de la fonction qui contient l'instruction retourner

fin

- Les paramètres de la définition d'une fonction (appelés formels) sont :
  - typés
  - séparés par ',' s'il y en a plusieurs
- Les paramètres formels sont utilisés pour ne pas lire les données dans une fonction.
- Les variables déclarées dans une fonctions (y compris les paramètres formels) sont appelées variables locales.

- Exemple: factoriel n en pseudo code:

```
Fonction fact(n : entier) : entier
  m, i : entier;
début
    m  := 1;
    pour i :=2 à n faire
            m := m * i;
    fpour
    retourner(m);
fin
```

- n! en C :

```
unsigned  int fact (unsigned int n) {
  unsigned int  i, m;
  m = 1;
  for (i = 2; i<= n; i++)
      m = m * i;
  return m;
}
```

- L'**appel** d'une fonction est utilisé dans une instruction sous la forme :

  <nom de la fonction> (liste des paramètres effectifs)

- Les paramètres formels et effectifs doivent correspondre en nombre et en type
- Lors d'un appel :
  - les paramètres formels reçoivent les valeurs des paramètres effectifs correspondant
  - le corps de la fonction est exécuté jusqu'au premier retourner rencontré
  - l'exécution se poursuit (à l'dresse de retour) dans la fonction appelante.
- Remarque: le type de retour d'une fonction peut être vide, et dans ce cas on écrit retourner() ou pas d'instruction retourner;

- Exemple: calcul de 1+2+…+n (n entier≥ 1 fixé)

```
Programme  somme_des_n_premiersTermes
// partie déclaration
n : entier; r : entier;   (ou n, r : entier)
Début
// Lecture des données
    Écrire (" n = ? " );  lire(n);
// appel et utilisation de la fonction
    sommeArith
    r := sommeArith(n);
    // affichage du résultat
    écrire("1+ 2 + … + n =  ", r);
fin
```

- Fonction pour calculer la somme 1+2+…+m

```
Fonction sommeArith ( m : entier) : entier
i, s  : entier;
début
    s := 0;
    pour i := 1 à m faire
            s := s + i;
    fpour;
 retourner(s);
fin
```

- **Deux types de passages des paramètres :**

  - Passage par valeur: la fonction travaille sur une copie du paramètre effectif transmis; i.e. la valeur du paramètre effectif n'est pas modifiée après l'appel de la fonction.

  - Passage par adresse (ou par référence):

  - l'identificateur du paramètre formel est précédé par le mot **ref**.
  - la fonction travaille directement sur l'identificateur du paramètre effectif; i.e. toute modification sur le formel entraine la même modification sur le paramètre effectif correspondant.

- Passage par valeur

```
Fonction échnger(x: réel, y : réel) : vide
z : réel;
début
      z := x;
      x := y;
      y := z;
fin

Fonction appelante()
a, b : réel;
début
      a := 2;  b:= 7;
      échanger(a,b);
      écrire( a = , a);
      écrire(  b = , b);
fin
```

Les résultas affichés par la fonction appelante :

a = 2 **b = 7**

- Passage par réference

```
Fonction echnger(ref x: réel,  ref y : réel) : vide
z : réel;
début
      z := x;
      x := y;
      y := z;
fin

Fonction appelante()
a, b : réel;
début
      a := 2;  b:= 7;
      échanger(a,b);
      écrire( a = , a);
      écrire(  b = , b);
fin
```

Les résultas affichés par la fonction appelante :

a = 7 **b = 2**

- Un **algorithme** se comporte comme une fonction sauf que l'on ne s'occupe pas des déclarations des variables ni de leurs types.

- Schéma d'un algorithme :

```
<nom de l'algorithme>(Liste des paramètres)

        début
        //bloc d'instructions
        fin
```

Ou

```
Algorithme <nom de l'algorithme>
    Données : // les variables qui sont des données de l'algo.
    Résultat(s): // variable(s) contenant le(s) résultat(s)

        début
        //bloc d'instructions
        //ne contenant pas retourner
        fin
```

On omet la partie déclaration des variables locales en adoptant la règle: les variables simples sont en minuscule et les tableaux en majuscule.

- **Algorithme pour calculer n!**

```
factoriel(n)
// n ≥ 0
début
      m := 1; i := 1;
      tantque i < n faire
            i := i + 1;
            m := m * i;
      ftantque
retourner(m);
fin
```

TABLEAUX STATIQUES

- TABLEAUX
- Un tableau est utilisé pour représenter une suite d'éléments de même type ($T = (t_1, t_2, \ldots, t_n)$).
- Déclaration d'un tableau (à un dimension):

<nom du tableau> : tableau [1 .. max] de <type des éléments>

(exemple T : tableau [1..20] de réel)

où : - max est une constante entière (positive)

- le type des éléments est quelconque (de base ou déclaré).

- La taille (ou longueur) d'un tableau est le nombre d'éléments du tableau; elle est toujours inférieure ou égale à max.

- les tableaux, en algorithmique, commencent à l'indice 1(en C et en java, ils commencent à l'indice 0)

TABLEAUX STATIQUES

- Opérations (de base) sur les tableaux:
- Accès à un élément du tableau :

  <nom du tableau> [<indice>]

  (indice est une expression de type entier)
- Parcours : (on utilise un indice et une boucle pour ou tant que)

  (exemple :

  pour i=1 à n faire //n est le nombre d'éléments du tableau T

  traiter(T[i]) // traiter() est une fonction ou algorithme à définir

  fpour;
- Recherche d'un élément dans un tableau
- Insertion d'un élément
- Suppression d'un élément

TABLEAUX STATIQUES

- Exemple : recherche de la position d'un élément dans un tableau de réels.

```
fonction localiser(T: tableau[1..max]de réel, n : entier, val :réel) : entier
    i : entier;
    trouve : booléen;
    début
            i := 1; trouve := faux;
            tantque (i ≤ n) et (non trouve) faire
                        si (T[i] = val) alors
                            trouve := vrai;
                        sinon  i := i + 1;
                        fsi;
            ftantque;
             si (trouve) alors retourner(i)
            sinon retourner(0);
            fsi
    fin
```

- insertion d'élément x dans un tableau T à n éléments à la position p.

```
Fonction inserer(T : tableau[1..max] de réel, ref n : entier, x : réel, p : entier) : vide
// 1 ≤p ≤ n
i : entier;
début
    i := n;
    tantque i ≥ p faire
            T[i+1] := T[i];
            i := i – 1;
    ftantque
    T[p] := x;
    n := n + 1;
fin
```

- Suppression d'une valeur d'un tableau T, qui se trouve à la position p.

```
Fonction supprimer(T : tableau[1.. max] de réel, ref n : entier, p : entier) : vide
i : entier;
début
    i := p;
    tantque i < n faire
        T[i] := T[i+1];
        i := i + 1;
    ftantque;
    n := n – 1;
fin
```

- Remarques :
  - ✓ Le nom du tableau, dans une liste de paramètres formels, est une référence.

    (en C, l'adresse de T[i], notée T + i, est calculée:

    adresse(T[i]) = adresse(T[0]) + sizeof(<type des éléments>) * i)
  - ✓ L'insertion (resp. suppression) d'un élément à un indice i, nécessite un décalage des (n- i +1) derniers éléments d'une position à droite (resp. à gauche)

STRUCTURE (ENREGISTREMENT)

- Structures (ou enregistrements)
- Le type structure est utilisé pour représenter une suite d'éléments pas nécessairement de même type, chaque élément est appelé champs.
- Déclaration d'un type structure:

<nom de type structure> = structure

<variable_champs1> : <type_champs1>;

<variable_champs2> : <type_champs2>;

.

.

fstructure

- Déclaration d'une variable de type structure:

<variable_structure> : <nom de type structure>

STRUCTURE (ENREGISTREMENT)

- Par analogie au type struct de C:
- Déclaration :

  struct <nom de la structure> {

  <type_champs1> <variable_champs1>

  <type_champs2> <variable_champs2>

  .

  .

  }
- Déclaration d'une variable de type structure:

  struct <nom de la structure> <variable>

- Ou avec la définition de type : **typedef**

  typedef struct <nom de la structure> <type_structure>;

  Déclaration d'une variable :

  <type_structure> <variable>

STRUCTURE (ENREGISTREMENT)

- Exemples :

```
Adresse = structure
        numero_rue : entier;
         nom_rue : tableau[1..20] de caractère;
         code_postal : entier;
fstructure

Point = structure
      abscisse : réel;
      ordonnee : réel;
fstructure

Déclaration des variables:
adr : Adresse;
p : Point
```

- En C :

```
struct adresse {
      int  numero_rue;
      char[20]  nom_rue;
      int  code_postal;
}
typedef struct adresse  Adresse;

typedef struct point {
      float  abscisse,  ordonnee;
}Point;

//Déclaration des variables:
Adresse  adr;
Point  p;
```

STRUCTURE (ENREGISTREMENT)

- Opération sur les structures :
- Accès à un champs :

  <variable de type structure>. <variable_champs>

  (ex: p.abcisse désigne l'abscisse du poin p)
- Affectation :

  <variable_type_structure> := <variable_type_structure>

  (L'affectation se fait champs par champs)

  Exemple : manipulation des complexes

STRUCTURE (ENREGISTREMENT)

- Déclaration de type complexe:

```
Complexe = structure
                p_reel, p_imag : réel;
fstructure

fonction plus(z1 : Complexe, z2 : Complexe) : Complexe
  z : Complexe;
debut
     z.p_reel  := z1.p_reel + z2.p_reel;
     z.p_imag  := z1.p_imag + z2.p_imag;
     retourner(z);
fin
```

- Déclaration de type complexe en C:

```
typedef struct complexe {
     double p_reel, p_imag ;
} Complexe;

Complexe  plus(Complexe z1,Complexe z2){
  Complexe  z;
   z.p_reel = z1.p_reel + z2.p_reel;
  z.p_imag = z1.p_imag + z2.p_imag;
  retun z;
}
```

ALGORITHMIQUE II

NOTION DE COMPLEXITE

- Comment choisir entre différents algorithmes pour résoudre un même problème?

- Plusieurs critères de choix :
  - Exactitude
  - Simplicité
  - **Efficacité** (but de ce chapitre)

- L'évaluation de la complexité d'un algorithme se fait par l'analyse relative à deux ressources de l'ordinateur:
  - Le temps de calcul
  - L'espace mémoire, utilisé par un programme, pour transformer les données du problème en un ensemble de résultats.

L'analyse de la complexité consiste à mesurer ces deux grandeurs pour choisir l'algorithme le mieux adapté pour résoudre un problème.(le plus rapide, le moins gourment en place mémoire)

On ne s'intéresse, ici, qu'à la **complexité temporelle** c.à d. qu'au temps de calcul (par opposition à la complexité spatiale)

- Le temps d'exécution est difficile à prévoir, il peut être affecté par plusieurs facteurs:
- la machine
- la traduction (interprétation, compilation)
- l'environnement (partagé ou non)
- L'habileté du programmeur
- Structures de données utilisées

- Pour pallier à ces problèmes, une notion de complexité plus simple, mais efficace, a été définie pour un **modèle de machine** . Elle consiste à compter les **instructions de base** exécutées par l'algorithme. Elle est exprimée en fonction de la taille du problème à résoudre.

- Une instruction de base (ou élémentaire) est soit: une affectation, un test, une addition, une multiplication, modulo, ou partie entière.

- La complexité dépend de la taille des données de l'algorithme.
- Exemples :
  - Recherche d'une valeur dans un tableau

    → taille (= nombre d'éléments) du tableau)
  - Produit de deux matrices

    → dimension des matrices
  - Recherche d'un mot dans un texte

    → longueur du mot et celle du texte

On note généralement:

n la taille de données, T(n) le temps (ou le cout) de l'algorithme.

T(n) est une fonction de IN dans IR^+

- Dans certains cas, la complexité ne dépend pas seulement de la taille de la donnée du problème mais aussi de la donnée elle-même.

  ➔Toutes les données de même taille ne génèrent pas nécessairement le même temps d'exécution.

  → (Ex. la recherche d'une valeur dans un tableau dépend de la position de cette valeur dans le tableau)

- Une donnée particulière d'un algorithme est appelée **instance** du problème.
- On distingue trois mesures de complexité:

1. Complexité dans le meilleur cas

$$T_{Min}(n) = \min \{T(d) \; ; \; d \text{ une donnée de taille } n\}$$

2. Complexité dans le pire cas

$$T_{Mx}(n) = \max \{T(d) \; ; \; d \text{ une donnée de taille } n\}$$

3. dans la cas moyen

$$T_{MOY}(n) = \sum_{d \text{ de taille } n} p(d).T(d)$$

$p(d)$: probabilité d'avoir la donnée d

$$T_{MIN}(n) \leq T_{MOY}(n) \leq T_{MAX}(n)$$

- Exemple. Complexité de la recherche d'un élément x dans un tableau A à n valeurs.

```
i := 1
tantque (i ≤n) et (A[i] ≠x) faire
            i := i+1;
ftantque
   si  i >n  alors retourner(faux);
   sinon retourner(vrai);
   fsi;
```

On note par:

a : le cout d'une affectation

t : cout d'un test

d : cout d'une addition

- **Cas le plus favorable**. x est le premier élément du tableau:

  $T_{min}(n) = 1a + 3t$

- **Pire des cas**. x n'est pas dans le tableau:

  $T_{max}(n) = (n+1)a + (2n+2)t + nd$

- **En moyenne :**

- **Complexité en moyenne**:
- On note par :

  D_i ($1 \leq i \leq n$) : ensemble de données (de taille n) où x est présent à la i^{eme} position

  D_{n+1} : ensemble de données où x n'est pas présent
- On suppose que la probabilité de présence de x dans une donnée est q. De plus, dans le cas où x est présent, on suppose que sa probabilité de présence dans l'une des positions est de 1/n

  On a :

  $p(D_i) = q/n$, $T(D_i) = i\,a + (2i+1)t + (i-1)d$; ($1 \leq i \leq n$)

  $p(D_{n+1}) = 1 - q$, $T(D_{n+1}) = T_{max} = (n+1)a + (2n+2)t + nd$

$p(D_i) = q/n$, $T(D_i) = i\,a + (2i+1)t + (i-1)d$; $(1 \leq i \leq n)$

$p(D_{n+1}) = 1 - q$, $T(D_{n+1}) = T_{max}(n) = (n+1)a+(2n+2)t+nd$

$$T_{MOY}(n) = \sum_{d \text{ de taille } n} p(d).T(d)$$

$p(d)$: probabilité d'avoir la donnée d

$$Tmoy(n) = \sum_{i=1}^{n} p(Di)T(Di) + p(Dn+1)T(Dn+1)$$

$$Tmoy(n) = \frac{q}{n}\sum_{i=1}^{n}(ia + (2i+1)t + (i-1)\,d) + (1-q)[(n+1)a + (2n+2)t + nd]$$

$$Tmoy(n) - q\,[\frac{n+1}{2}a + (n+2)t + \frac{n-1}{2}d] + (1-q)[(n+1)a + (2n+2)t + nd]$$

Cas où q=1, i.e. x est toujours présent: $Tmoy(n) = {}^{1}/_{2}\,[(n+1)a + (2n+4)t + (n-1)d]$

$$= \frac{n}{2}(a + 2t + d) + {}^{1}/_{2}\,(a + 4t - d)$$

On remarque que la complexité de cet algo. est de la forme: $\alpha\, n + \beta$, α et β son des constates

Complexité asymptotique
Comportement de T(n)

- Pour mesurer la complexité d'un algorithme, il ne s'agit pas de faire un décompte exact du nombre d'opérations T(n), mais plutôt de donner un ordre de grandeur de ce nombre pour n assez grand.
- **Notation de Landau**
- **"grand O"**

  $T(n) = O(f(n))$ ssi

  $\exists\, c > 0\ \exists\, n_0 > 0\ \forall\, n > n_0\quad T(n) \leq c.f(n)$
- **"grand oméga"**

  $T(n) = \Omega\,(f(n))$ ssi

  $\exists\, c > 0\ \exists\, n_0 > 0\ \forall\, n > n_0\ T(n) \geq\ c.f(n)$
- **"grand théta"**

  $T(n) = \Theta(f(n))$ ssi

  $\exists\, c_1 > 0\ \exists\, c_2 > 0\ \exists\, n_0 > 0\ \forall\, n > n_0\ \ c_1 f(n) \leq T(n) \leq c_2 f(n)$

Remarque: les constantes c, c_1, c_2 et n_0 sont indépendantes de n

- D'une manière générale, $f : IR \to IR$
  - $f(x) = O(g(x))$ s'il existe un voisinage V de x_0 et une constante $k>0$ tels que $|f(x)| \leq k|g(x)|$, $(x \in V)$

- Si la fonction g ne s'annule pas, il revient au même de dire que le rapport $\left|\frac{f(x)}{g(x)}\right|$ est borné pour $x \in V$.

-Exemple: au voisinage de 0, on a:

$$x^2 = O(x), \quad \ln(1+x) = O(x)$$

- Au voisinage de l'infini (comme pour le cas de la complexité), il existe $a>0$ ($V =]a, +\infty[$) et $k>0$ t.q

$$|f(x)| \leq k|g(x)| , \quad \forall x > a$$

et on dit que f est dominée asymptotiquement par g.

(Au voisinage de $+\infty$, on a: $x = O(x^2)$, $\ln x = O(x)$)

- **Remarques**

1. $f = O(g) \Leftrightarrow$ le quotient $\left|\frac{f(x)}{g(x)}\right|$ est bornée au voisinage de l'infini.
2. $\lim_{x \to \infty} \left|\frac{f(x)}{g(x)}\right| = a \rightarrow f = O(g)$
3. si $a = 1$, on écrit $f \sim g$ et on a $f = \Theta(g)$.
4. $f = \Theta(g)$ ne signifie pas que le quotient $f(x)/g(x)$ tend vers une limite (1 en particulier).

Exemple. $f(x) = x(2+\sin x)$. On a $x \leq f(x) \leq 3x$ $\forall x>0$, donc $f(x) = \Theta(x)$. En revanche, le quotient $f(x)/x$ ne tend vers aucune limite lorsque $x \rightarrow +\infty$

- Abus de notation:

  On a par définition:

  $O(f) = \{g: IN \to IR \; / \; \exists\, c > 0 \;\; \exists\, n_0 > 0 \quad g(n) \leq c.f(n) \; , \forall\, n > n_0\}$

  $\Omega(f) = \{g: IN \to IR \; / \; \exists\, c > 0 \;\; \exists\, n_0 > 0 \quad g(n) \geq c.f(n) \; , \forall\, n > n_0\}$

  $\Theta(f) = O(f) \cap \Omega(f)$

La difficulté, dans la familiarisation avec ces concepts, provient de la convention de notation (de Landau) qui veut que l'on écrive :

$g = O(f)$, ou encore $g(n) = O(f(n))$ au lieu de $g \in O(f)$

De manière analogue, on écrit $O(f) = O(g)$ lorsque $O(f) \subset O(g)$

(il en est de même pour les notations Ω ou Θ)

- Exemple. Soit la fonction $T(n) = \frac{1}{2} n^2 + 3n$
  - $T(n) = \Omega(n)$ ($n_0 = 1$, $c = \frac{1}{2}$)
  - $T(n) = \Theta(n^2)$ ($n_0 = 1$, $c_1 = \frac{1}{2}$, $c_2 = 4$)
  - $T(n) = O(n^3)$ ($n_0 = 1$, $c = 4$)
- $T(n) \neq O(n)$

Supposons que $T(n) = O(n)$

$\exists\ c > 0, \exists n_0 > 0 : \frac{1}{2} n^2 + 3n \leq cn\ \forall n \geq n_0$

donc $c \geq \frac{1}{2} n$, contradiction. (la constante c ne peut dépendre de n)

- Remarques

1. Si T(n) est un polynôme de degré k alors $T(n) = \Theta(n^k)$
2. $O(n^k) \subset O(n^{k+1})$ $\forall K \geq 0$ (idem pour Θ)
3. $\Theta(f(n)) \subset O(f(n))$ pour toute fonction f positive
4. $O(1)$ utilisé pour signifier « en temps constant »

- Remarques pratiques:
  - Le cas le plus défavorable est souvent utilisé pour analyser un algorithme.
  - La notation O donne une borne supérieure de la complexité pour toutes les données de même taille(suffisamment grande). Elle est utilisée pour évaluer un algorithme dans le cas le plus défavorable.
  - $T(n) \leq cf(n)$ signifie que le nombre d'opérations ne peut dépasser $cf(n)$ itérations, pour n'importe quelle donnée de longueur n.
  - Pour évaluer la complexité d'un algorithme, on cherche un majorant du nombre d'opérations les plus dominantes.
  - Dans les notations asymptotiques, on ignore les constantes.

L'algorithme de recherche dans un tableau à n éléments, cité précédemment, est en O(n)

Ordre de grandeur courant

- $O(1)$: complexité constante
- $O(\log(n))$: complexité logarithmique
- $O(n)$: complexité linéaire
- $O(n^2)$: complexité quadratique
- $O(n^3)$: complexité cubique
- $O(2^n)$: complexité exponentielle

- Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme.

  On suppose que l'ordinateur utilisé peut effectuer 10^6 opérations à la seconde (une opération est de l'ordre de la µs)

n\T(n)	log n	n	n log n	n^2	2^n
10	3 µs	10 µs	30 µs	100 µs	1000µs
100	7 µs	100 µs	700 µs	1/100 s	10^{14} siècles
1000	10 µs	1000µs	1/100µs	1 s	astronomique
10000	13 µs	1/100µs	1/7 s	1,7 mn	astronomique
100000	17 µs	1/10 s	2 s	2,8 h	astronomique

•

- Un algorithme est dit **polynomial** si sa complexité est en $O(n^p)$.
- Un algorithme est dit praticable s'il est polynomial ($p \leq 3$).

 Les algorithmes polynomiaux où $p > 3$ sont considérés comme très lents

 (un algorithme polynomial de l'ordre de n^5 prendrait environ 30 ans pour n=1000)
- Un algorithme est dit exponentiel si sa complexité est supérieure à tout polynôme.

- Deux grandes classes de la complexité :
  - $\mathbb{P}$ classe des algorithmes polynomiaux
  - $\aleph\ \mathbb{P}$ classe des algorithmes « Non déterministe polynomiale »

 On a :

$$(O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(e^n) \subset O(n!)$$

- **Propriétés**

  En utilisant la notation de Landau (pour les fonctions de IN dans IR^+), on a :

1. $f + O(g) = \{f + h \ / \ h \in O(g)\}$

   $(h = f + O(g) \Leftrightarrow h - f \in O(g)$.

   $f\ O(g) = \{f\ h \ / \ h \in O(g)\}$
2. $f = O(f)$
3. $f = O(g),\ g = O(h) \Rightarrow f = O(h)$
4. $c\ O(f) = O(c\ f) = O(f)$ ($c>0$)
5. $O(f) + O(g) = O(f + g) = O(\max(f, g))$
6. $O(f) + O(f) = O(f)$
7. $O(f)\ O(g) = O(fg)$
8. $f = O(g) \Leftrightarrow g = \Omega(f)$

Calcul de la complexité: règles pratiques

1. la complexité d'une suite d'instructions est la somme des complexités de chacune d'elles.

2. Les opérations élémentaires telle que l'affectation, test, accès à un tableau, opérations logiques et arithmétiques, lecture ou écriture d'une variable simple ... etc, sont en O(1).

3. T(si C alors A1 sinon A2) = max(T(C),max(T(A1),T(A2)))

4. T(pour i:=e_1 à e_2 faire Ai fpour) = $\sum_{i=e1}^{e2} T(Ai)$

 (si Ai ne contient pas de boucle dépendante de i et si Ai est de complexité O(m) alors la complexité de cette boucle « pour » est $O((e_2 - e_1 + 1)m)$.)

5. La difficulté, pour la boucle tantque, est de déterminer le nombre d'itération Nb_iter (ou donner une borne supérieure de ce nombre)

 T(tantque C faire A ftantque) = O(Nb_iter x (T(C) + T(A))

- Exemples

1. **Calcul de la somme 1+2+…+n**

```
S:=0;  //O( 1)
Pour i:=1  à n faire
   s:= s + i; // O(1)
fpour;
```

$\sum_{1}^{n} O(1)$

O(n)

O(1) + O(n) = O(n)

T(n) = O(n)

2. **Calcul de :** $T[i] = \sum_{j=1}^{i} j \quad pour\ i = 1, 2, \dots, n$

```
pour i := 1  à n faire
   s := 0; // O(1)
   pour j := 1 à i faire
        s := s + j; // O(1)
   fpour;
   T[i] := s; //O(1)
fpour;
```

O(i)

O(1)+O(i)+O(1)=O(i)

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

$= O(n^2)$

$T(n) = O(n^2)$

3. **Analyse de l'algo. Suivant :**

```
Donnée  n; (n>0)
Résultat  cpt;

début
cpt := 1;
tantque   n ≥ 2  faire
    n := n div 2;
    cpt := cpt +1;
ftantque
fin
```

Que calcule cet algo?
Quelle est sa complexité?

- cpt = 1 + le nombre d'itérations
- Le nombre d'itérations = nombre de division de n par 2.
- Soit p ce nombre.

  - si n est une puissance de 2, i.e. $n = 2^p$ alors $p=\log_2(n)$.

  - p vérifie: $2^p \leq n < 2^{p+1}$

  $p \leq \log_2(n) < p+1 \Rightarrow p=E(\log_2(n))$
- cpt = 1 + E(log2(n)), cette expression de cpt correspond au nombre de bits nécessaires pour représenter l'entier n.
- $T(n) = O(\log(n))$

ALGORITHMIQUE II

TRIS ITERATIFS

- Le problème:

  Etant donnée une suite de n nombres $(a_1,a_2,\ldots, a_n)$, on cherche une permutation (arrangement) des éléments de cette suite $(a'_1,a'_2,\ldots, a'_n)$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

  ♦ A partir de la suite (7,1,2,6), un algorithme de tri donne comme résultat la suite (1,2,6,7).

- On se limite aux nombres entiers rangés dans un tableau A à n éléments.
- Dans le cas où les éléments sont des collections de données (enregistrement), on trie le tableau suivant une **clé** (champ de l'enregistrement).

Tri par sélection:

<u>Algorithme</u>:

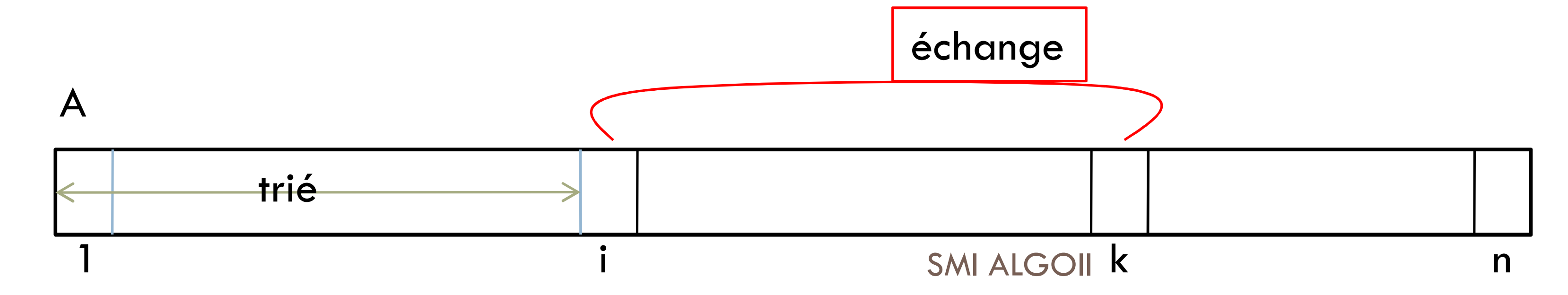
pour i=1 à n-1 faire

-chercher le 1ème minimum, A_k , de $\{A_{i+1},\ldots,A_n\}$

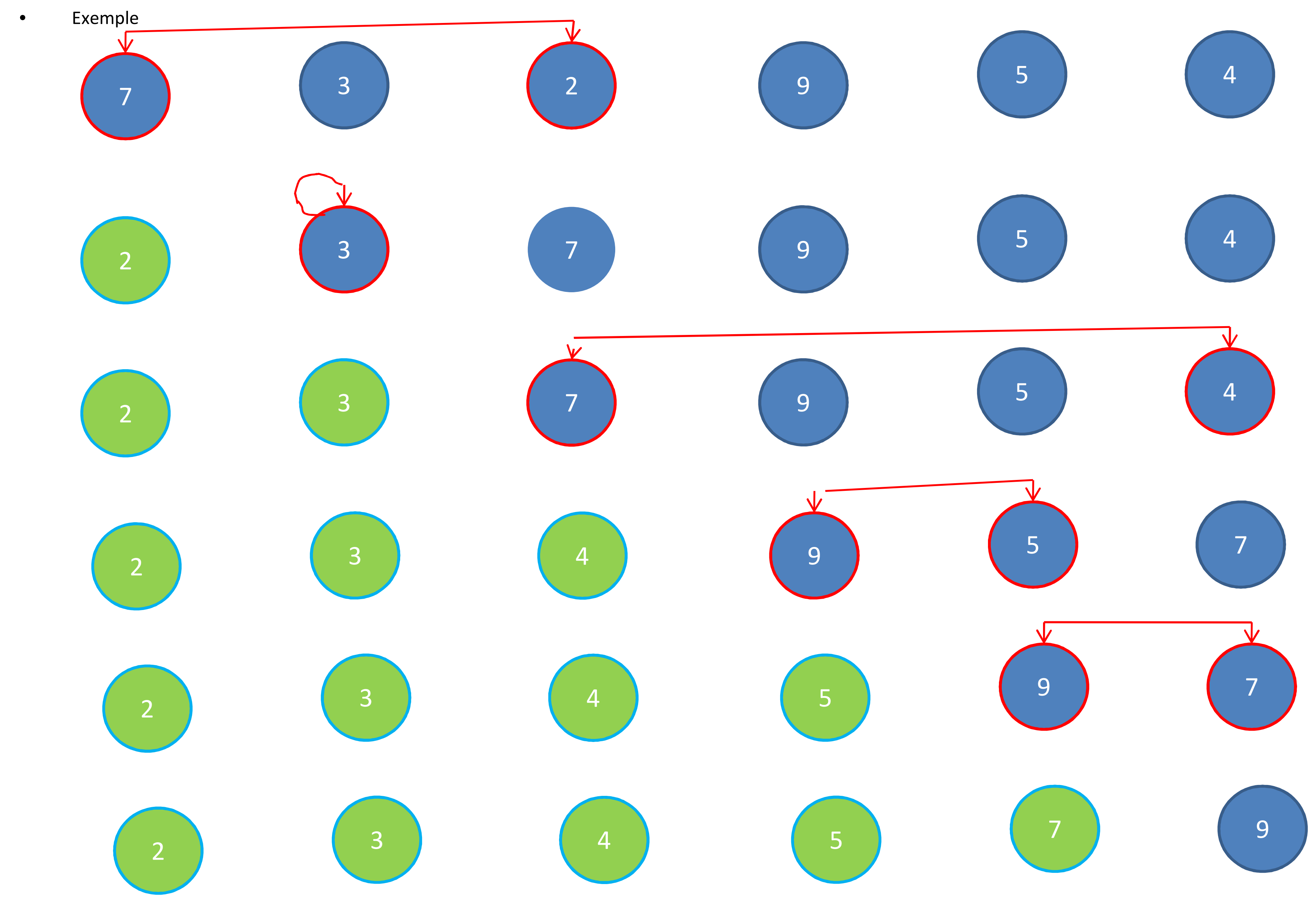
(K est l'indice de min$\{A_{i+1},\ldots,A_n\}$ dans le tableau A)

- échanger A_i et A_k

L'algorithme fonctionne selon le schéma suivant:



- Exemple



Analyse du tri par sélection

- Algorithme:

Tri_selection(A,n)

début

```
pour i := 1 à n-1 faire
    //Recherche de min{A_i, …, A_n} = A_k
    k := i;
    pour j:=i+1 à n faire
            si A[j] < A[k] alors k:=j;
            fsi;
    fpour
    // échange de A_k et A_i
    temp := A[k];
    A[k] := A[i];
    A[i] := temp;
fpour;
fin
```

- La boucle j détermine le iè minimum; elle tourne n-i fois (au maximum) pour faire (n-i) tests d'éléments.
- Les échanges de A[k] et A[i] demandent 3 opérations.
- La complexité du corps de la boucle j est de la forme

  a(n-i)+b, donc en O(n-i) (i=1, …, n-1).
- La complexité de l'algorithme est de l'ordre de:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$T(n) = O(n^2)$

TRI par ISERTION

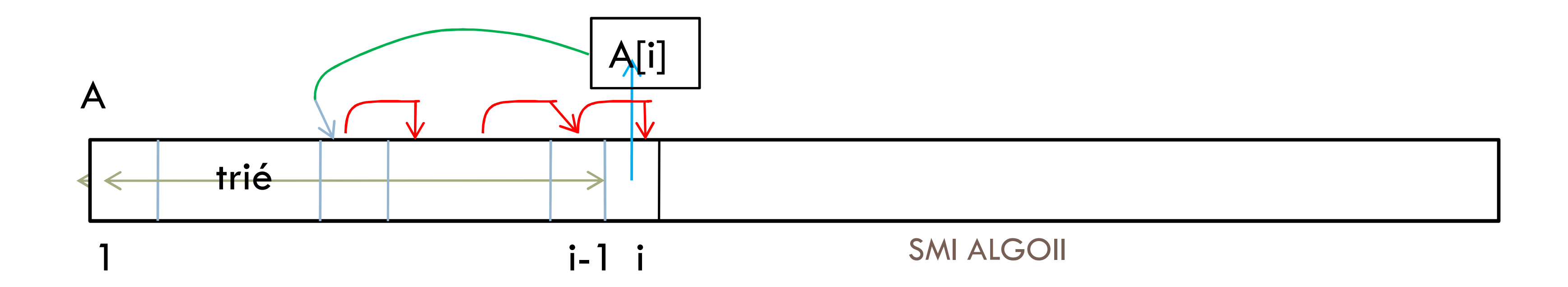
- Algorithme:

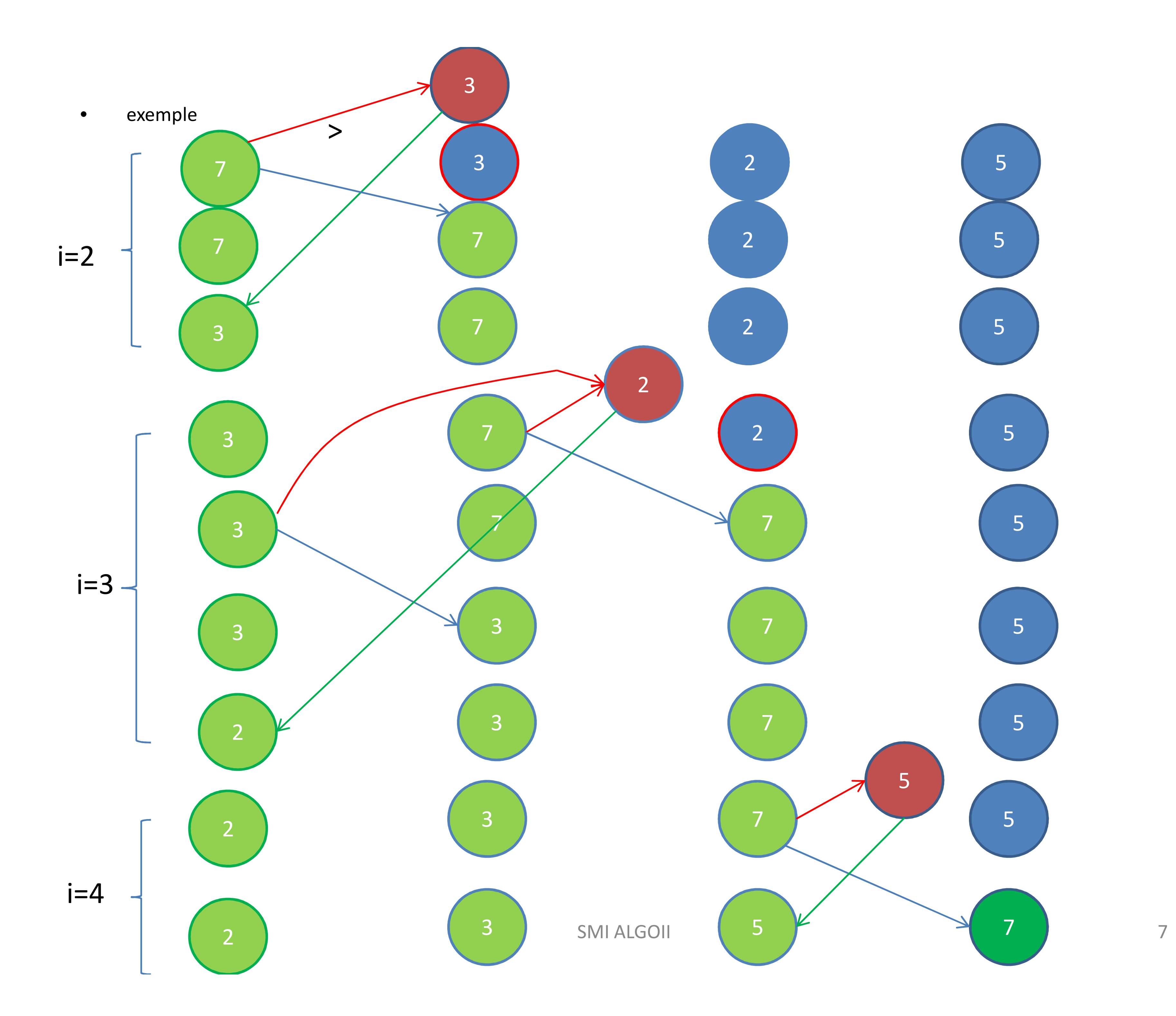
Pour i := 2 à n faire

- on insère A[i], à sa place, dans le sous-tableau A[1..i-1]

(On cherche le 1[er] élément $\leq$ A[i] parmi {A[i-1],…,A[1]} en décalant d'une position à droite)

L'algorithme fonctionne selon le schéma suivant:




• exemple
>
i=2
i=3
i=4
3
3 2 5
7 7 2 5
7 2 5
3 7 2 5
2
3 7 2 5
3 7 7 5
3 3 7 5
2 3 7 5
5
2 3 7 5
2 3 5 7

Analyse du tri par insertion

- <u>Algorithme</u>:

Tri_Insersion(A,n)

début

pour i := 2 **à** n **faire**

cle := A[i];

j := i-1;

Tantque (j≥1) et (cle<A[j]) **faire**

A[j+1] := A[j];

j := j-1;

ftantque

A[j+1] = cle;

fpour

fin

- La boucle j tourne, dans le pire des cas, (i-1) fois

  (i-1 comparaisons et i-1 décalages)

- La complexité du corps de la boucle i est de la forme

  a(i-1)+b, pour i=2,…,n.

- La complexité de l'algorithme:

$$\sum_{i=2}^{n} a(i-1)+b = \sum_{k=1}^{n-1} ak+b = a\frac{n(n-1)}{2} + b(n-1)$$

$T(n) = O(n^2)$

Tri à Bulles

- On dit qu'on a une inversion s'il existe (i,j) tels que i<j et $a_i > a_j$
- $(a_1,\ldots,a_i, a_{i+1},\ldots,a_n) \longrightarrow (a_1,\ldots,a_{i+1},a_i,\ldots,a_n)$
- Un tableau est trié s'il n'a aucune inversion. La complexité du tri est proportionnelle au nombre d'inversions qui est de l'ordre de C_n^2 (nombre de couple(i,j) tels que i<j).
- L'algorithme consiste à parcourir le tableau à trier en examinant si chaque couple d'éléments consécutifs (a_i, a_{i+1}) est dans le bon ordre ou non, si ce couple n'est pas dans le bon ordre on échange ses éléments et ce processus est répété tant qu'il reste des inversions à faire.

Tri à Bulles

- Algorithme1:

```
Bulles1(A,n)
Début
    fini := faux;
    Tant que non fini faire
     i : = 1 ;
        tant que (i < n) et (A[i] ≤ A[i+1] ) faire
                i := i + 1 ;
       ftantque;
        si  i < n  alors
                échanger (A[i], A[i+1]) ;
                fini := faux ; }
        sinon
                fini := vrai ;
        fsi;
    ftantque;
fin
```

- On remarque que les transpositions successives font pousser le maximum à la dernière position du tableau si on fait un balayage de gauche à droite et le minimum à la 1ère position si on fait un balayage de droite à gauche.
- Algorithme2:

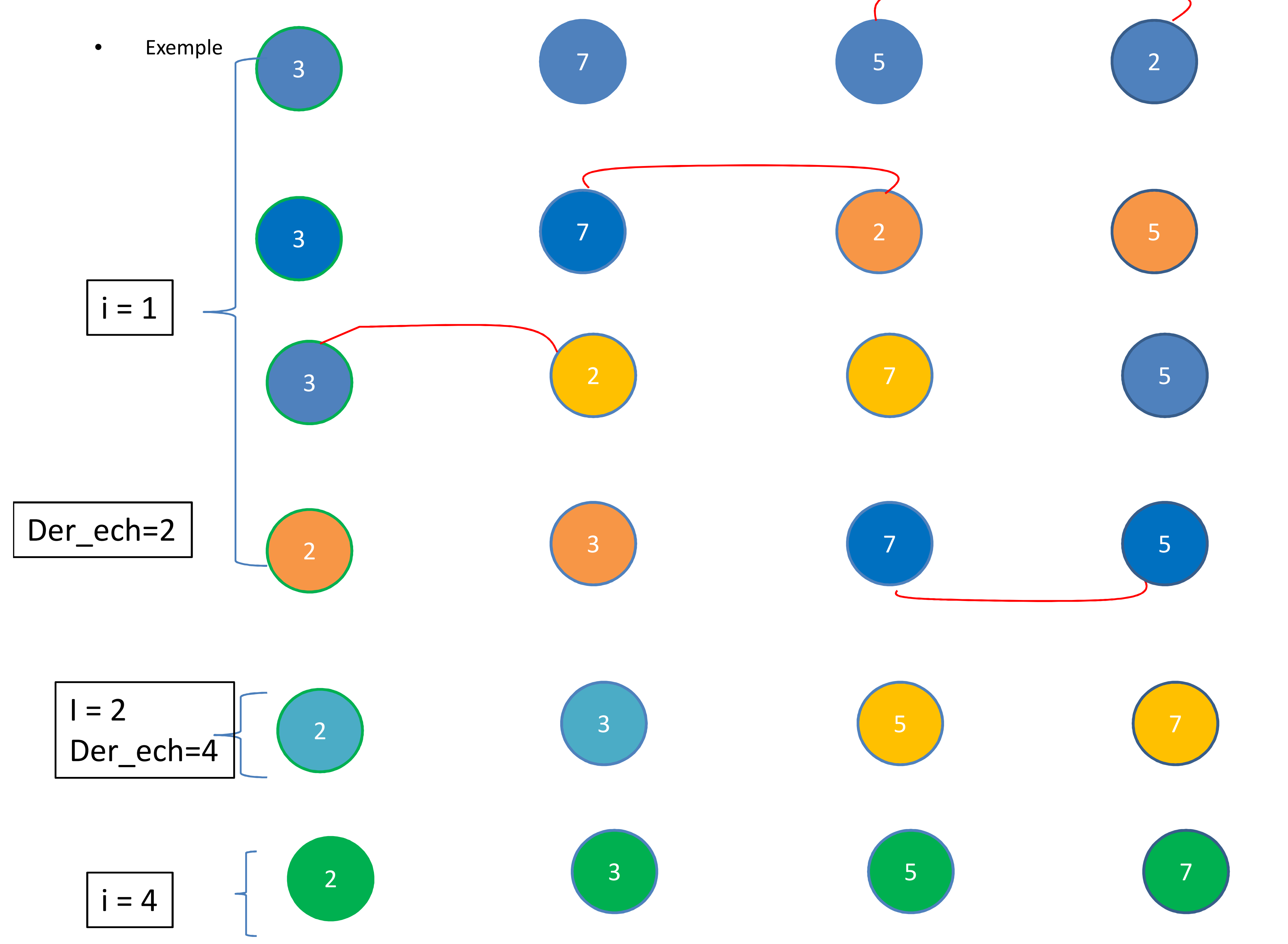
```
Bulles2(A,n)
Début    i := 1;
    tantque i ≤ n-1  faire  der_ech := n;
       pour j := n à i+1 pas -1  faire
           si A[j-1] > A[j] alors
            échanger(A[j], A[j-1]); der_ech := j;
           fsi;
       fpour; i := der_ech;
    ftantque;
Fin
```

Le nombre de comparaison d'éléments de A ≤ $\frac{n(n-1)}{2}$

Le nombre d'échanges ≤ $\frac{n(n-1)}{2}$

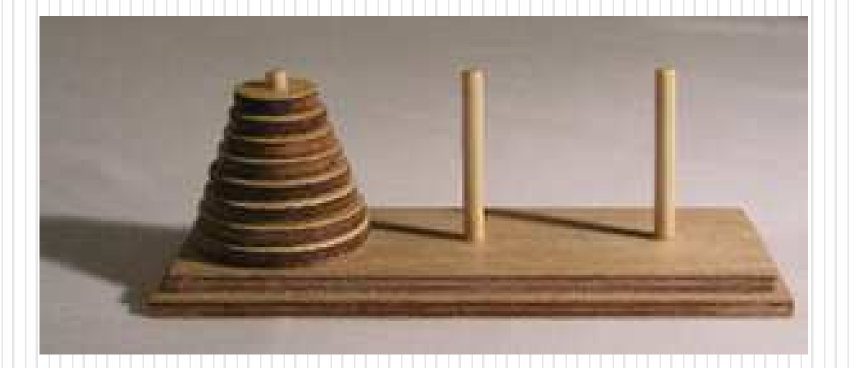
$T(n) = O(n^2)$

- Exemple



ALGORITHMIQUE II

Récurrence et Récursivité



Récurrence

- **Suite récurrente:** la définition d'une suite est la donnée
- d'un terme général défini en fonction du (ou des) terme(s) précédant(s)
- D'un terme initial qui permet d'initialiser le calcul
- **Principe de récurrence :**

Soit P un prédicat (ou propriété) sur IN qui peut être soit vrai soit faux (on écrira souvent P(n) à la place de P(n) = vrai).

On suppose que

- P(0) vrai
- $\forall n \in$ IN, P(n) $\Rightarrow$ P(n+1)

Alors , pour tout n $\in$ IN, P(n) est vrai.

Si on considère le prédicat suivant

P(n) : je sais résoudre le problème pour n

alors le principe de récurrence nous dit que si je sais résoudre le Pb pour n=0 et que si je sais exprimer la solution pour n en fonction de la solution pour n+1 alors je sais résoudre le Pb pour n'importe quel n.

Récursivité

- Examples:

1. Puissance

$$\begin{cases} a^0 = 1 \\ a^{n+1} = a\ a^n \end{cases}$$

Ou bien

$$\begin{cases} a^0 = 1 \\ a^n = a\ a^{n-1} \quad n>0 \end{cases}$$

2. Factoriel

$$\begin{cases} 0! = 1 \\ n! = n\ (n-1)! \quad ,\ n \geq 1 \end{cases}$$

3. Suite de Fibonacci

$$\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad ,\quad n \geq 2 \end{cases}$$

Récursivité

- Un algorithme (ou fonction) est dit récursif s'il est défini en fonction de lui-même.
- Exemples
  - Fonction puissance(x : réel, n : entier) : réel

```
début
        si  n = 0 alors retourner 1
        sinon retourner (x * puissance(x , n-1))
fin
```

  - Factoriel (n)

```
début
        si n = 0 alors retourner(1)
        sinon  retourner (n*factoriel(n-1))
fin
```

Récursivité

- *fact (n)*

  début

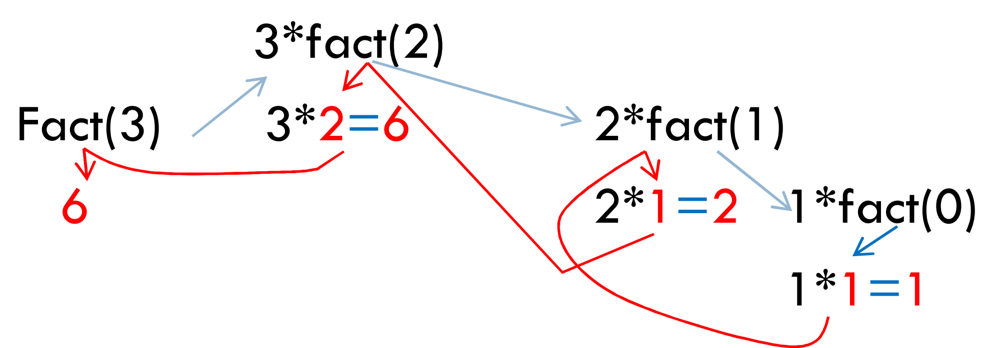
  si n = 0 alors retourner(1)

  *sinon retourner(n*fact(n-1))*

  fsi

  fin

- Le déroulement de l'appel de fact(3):



- La condition n = 0 est appelée test d'arrêt de la récursivité.
- Il est impératif de prévoir un test d'arrêt dans une fonction récursive, sinon l'exécution ne s'arrêtera jamais.
- L'appel récursif est traité comme n'importe appel de fonction.

Récursivité

- L'appel d'une fonction (récursive ou non) se fait dans un **contexte d'exécution** propre (pile d'exécution), qui contient :
  - L'adresse mémoire de l'instruction qui a appelé la fonction (adresse de retour)
  - Les valeurs des paramètres et des variables locales à la fonction.
- L'exécution d'une fonction récursive se fait par des appels successifs à la fonction jusqu'à ce que la condition d'arrêt soit vérifiée, et à chaque appel, les valeurs des paramètres et l'adresse de retour sont mis (empilés) dans la pile d'exécution.

Récursivité

- L'ordre des instructions par rapport à un appel récursif est important.
- Exemple:

afficher(n)

début

si n > 0 alors

1

afficher(n div 10)

2

fsi

fin

écrire(n mod 10)

- L'algorithme récursif afficher(n) permet d'afficher les chiffres d'un entier, strictement positif, selon la disposition de l'instruction écrie(n mod 10):

- Si l'instruction est placée en 1 , les chiffres sont affichés dans l'ordre inverse

- Si elle est placée en 2, alors les chiffres seront affichés dans le bon ordre

Pour n = 123, on a :

1 → 3 2 1

2 → 1 2 3

Type de récursivité

- Récursivité simple: Une fonction récursive contient un seul appel récursif.
- Récursivité multiple: une fonction récursive contient plus d'un appel récursif (exemple suite de Fibonacci).
- Récursivité mutuelle(ou croisée): Consiste à écrire des fonctions qui s'appellent l'une l'autre.

  Exemple

Récursivité mutuelle

```
Pair(n)
début
   si  n = 0 alors
         retourner vrai
   sinon
    retourner (impair(n-1))
   fsi
fin
```

```
Impair(n)
début
   si  n = 0 alors
         retourner (faux)
   sinon
      retourner (pair(n-1))
   fsi
fin
```

Un peu de Structures de Données

- **Notion de pile.**

  Une pile est une structure pour représenter une suite d'éléments avec la contrainte qu'on ne peut ajouter, ou enlever, un élément que d'un même côté de la pile (dit sommet de la pile).

- Exemple pile d'assiettes.
- Une pile peut être représentée

  par un tableau et un indice de sommet

ajouter

enlever

som

4

P

Notion de Pile

- **Opérations définies sur les piles:**
  - **initialiser(p : Pile)** //Crée une pile vide.
  - **sommet(p : Pile) : élément**// Renvoie l'élément au sommet de la pile p, sous la condition que p soit non vide.
  - **empiler(x : élément, p : Pile)** // ajoute x au sommet de la pile p.
  - **dépiler(p : Pile)** // supprime l'élément au sommet de la pile p, sous la condition que p soit non vide.
  - **pileVide(p : Pile) : booléen** // retourne vrai si p est vide.

Notion de Pile

- **Exemple.**

  Une expression e est dite bien parenthésée (on se limite au '(' et ')') si :

1. Le nombre de parenthèses ouvrantes ($|e|_($) est égal au nombre de parenthèses fermantes ($|e|_)$) dans e.
2. Tout préfixe (partie gauche) u de e vérifie: $|u|_(- |u|_(\geq 0$.

Algorithme: on parcourt l'expression e (de gauche à droite). A chaque rencontre d'une parenthèse ouvrante on l'empile, et à chaque rencontre d'une parenthèse fermente on dépile.

Si on arrive à la fin de e avec une pile vide, l'expression e est bien parenthésée sinon e n'est pas bien parenthésée.

- l'expression (()())() est bien parenthée.

- l'expression ())(n'est pas bien parenthésée.

Transformation du récursif en itératif : « Dérécursivation »

- **Schéma d'un algorithme récursif:**

```
algoR(X)
début
        A
    si C(X) alors
        B;
        algoR(φ(X));
        D;
    sinon
        E;
    fsi;
fin
```

Où :
X : liste de paramètres
C : condition d'arrêt portant sur X
A, B, D, E : bloc d'instructions (éventuellement vide)
$\varphi(X)$: transformation des paramètres

Transformation du récursif en itératif : « Dérécursivation »

```
algoR(X)
Début
        A
    si C(X) alors
        B ;
        algoR(φ(X));
        D ;
    sinon
        E ;
    fsi;
fin
```

- Algorithme itératif équivalent.

```
algoI(X)
p : Pile
début
    initialiser(p);
        A ;
    tantque C(X) faire
        B ;
        empiler(X, p);
        X := φ (X);
        A ;
    ftantque;
        E ;
    tantque (non pileVide(p))  faire
        X := sommet(p);
        dépiler(p);
        D ;
    ftantque
fin
```

Dérécusivation

- Exemple.

```
afficherR(n)
    début
        si  n > 0  alors
            afficher(n div 10)
            écrire(n mod 10);
        fsi
    fin
```

$A = B = E = \emptyset$

```
afficherI(n)
p : Pile;
Début
    initialiser(p);
    tanque  n > 0 faire
            empiler(n, p);
            n := n div 10;
    ftantque
    tantque (non pileVide(p)) faire
            n := sommet(p);
            dépiler(p);
            écrire(n mod 10);
    ftanque
fin
```

Transformation du récursif en itératif : « Dérécursivation »

- **Récursivité terminale:**

  La récursivité est dite <u>terminale</u> si la dernière instruction exécutée est un appel récursive; (Cas où D = ∅). Il est claire, dans ce cas, d'éliminer la pile dans la version itérative. (On dépile pour ne rien faire dans la 2ème boucle).

- La récursivité d'une fonction **F**(X) est aussi dite terminale lorsqu'elle se termine par l'instruction retourner(**F**(φ(X))). On ajoute, dans ce cas, un paramètre à la liste X pour contenir le résultat de retour d'un appel récursive, comme le montre l'exemple suivant:

Exemple

```
fonction FACR(n) ;
début
        si n=0 alors retourner (1)
        sinon retourner (n* FACR(n-1)) ;
fin.
```

RÉCURSIVE

```
fonction FACR'(n, r)
        début
            si n=0 alors retourner (r)
        sinon retourner (FACR'(n-1, n*r)) ;
        fin.
```

RÉCURSIVITÉ TERMINALE

```
fonction FACI(n) ;
début
    r:= 1 ;
    tant que n > 0 faire
    { r := n* r ; n := n-1 ; }
    retourner (r) ;
    fin.
```

ITÉRATIVE

```
fonction FACI'(n, r) ;
        début
                tant que n > 0 faire
        { r := n* r ; n := n-1 ; }
        retourner (r) ;
        fin.
```

Complexité des algorithmes récursifs

La complexité des algorithmes récursifs est souvent exprimée par une équation de récurrence.

- Exemple1. Complexité de l'algorithme récursif pour calculer n! (l'opération dominante est la multiplication)

  Soit T(n) le coût de FACR(n). T(n) vérifie l'équation:

$$\begin{cases} T(0) = 0 \\ T(n) = T(n-1) + 1 \end{cases}$$

  la solution de cette équation est :

  $T(n) = n = 1 + 1 + \ldots + 1$ (n fois)

Complexité des algorithmes récursifs

- Exemple2.
- Tri par sélection

```
 sel_rec(T,n)
début
  si n > 1 alors
        k← max {i ∈{1,2,…,n} /  T[i] ≥ T[j] , j=1,2,…,n et j ≠ i}
        échanger(T[k],T[n])
        sel_rec(T,n-1)
  fsi;
fin
```

- Complexité : T(n) vérifie :

$$\begin{cases} T(1) = 0 \\ T(n) = T(n-1) + n \end{cases}$$

$T(n) = n+T(n-1) = n+(n-1)+T(n-2) =\ldots= n+(n-1)+\ldots+2 +C(1) = \frac{n(n+1)}{2} - 1$

$T(n) = O(n^2)$

On ouvre une parenthèse

Encore un peu de structures de données

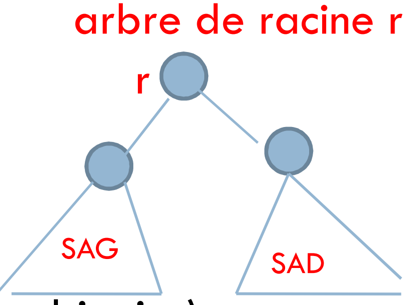
- **Notion d'arbre binaire**

  On introduit cette notion pour savoir interpréter les arbres d'appels dans le cas d'une récursivité double(où il y a deux appels récursifs).

  - Les arbres sont utilisés pour représenter une suite d'éléments.

  - un arbre est un ensemble de nœuds, chaque nœud représente un élément de la suite.

  Définition récursive d'un arbre binaire:



- un arbre binaire est:
  - Soit vide
  - Soit formé :
    - D'un nœud (appelé racine)
    - D'un sous-arbre gauche , noté SAG, qui est un arbre binaire)
    - D'un sous-arbre droit, noté SAD, qui est aussi un arbre binaire

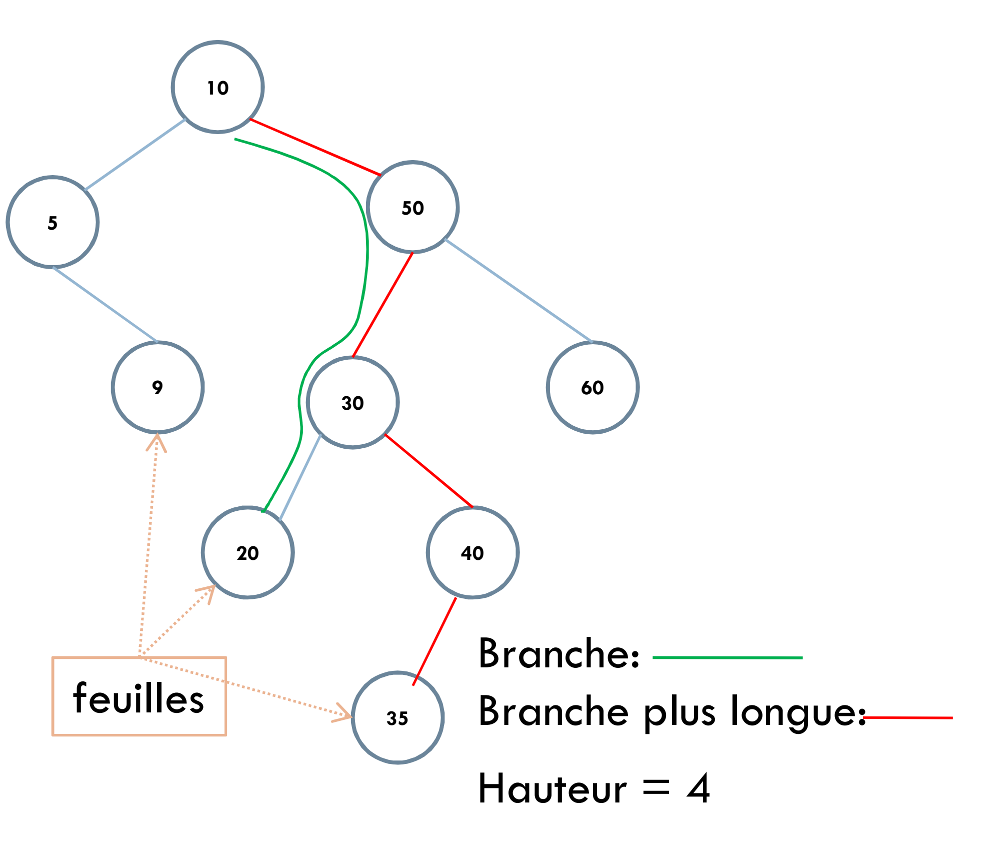
(les deux sou-arbres gauche et droit sont disjoints).

Arbre binaire: terminologie

Soit A un arbre binaire de racine r.

- La racine du SAG (resp. SAD) de A est appelé fils gauche (resp. fils droit) de r et r est appelé père.
- Un nœud qui n' a pas de fils est appelé feuille.
- un chemin de l'arbre est une suite de nœuds $n_1, n_2, \ldots n_k$ où n_{i+1} est un fils (gauche ou droit) de n_i , $1 \leq i \leq k-1$.
- Longueur d'un chemin = nombre de nœuds, constituant le chemin, - 1
- Une branche est un chemin de la racine à une feuille.
- La hauteur d'un arbre est la longueur de la plus longue branche de l'arbre.

$$h(A) = \begin{cases} -1 \text{ si } A = \emptyset \\ 1 + \max(h(SAG(A)), h(SAD(A)) \end{cases}$$



Arbre binaire

- Résultat (utile pour la complexité sur les arbres binaires de recherche):

la hauteur h d'un arbre binaire de taille n (n est le nombre de nœuds de l'arbre) vérifie: $1 + \lceil \log_2 n \rceil \leq h \leq n$

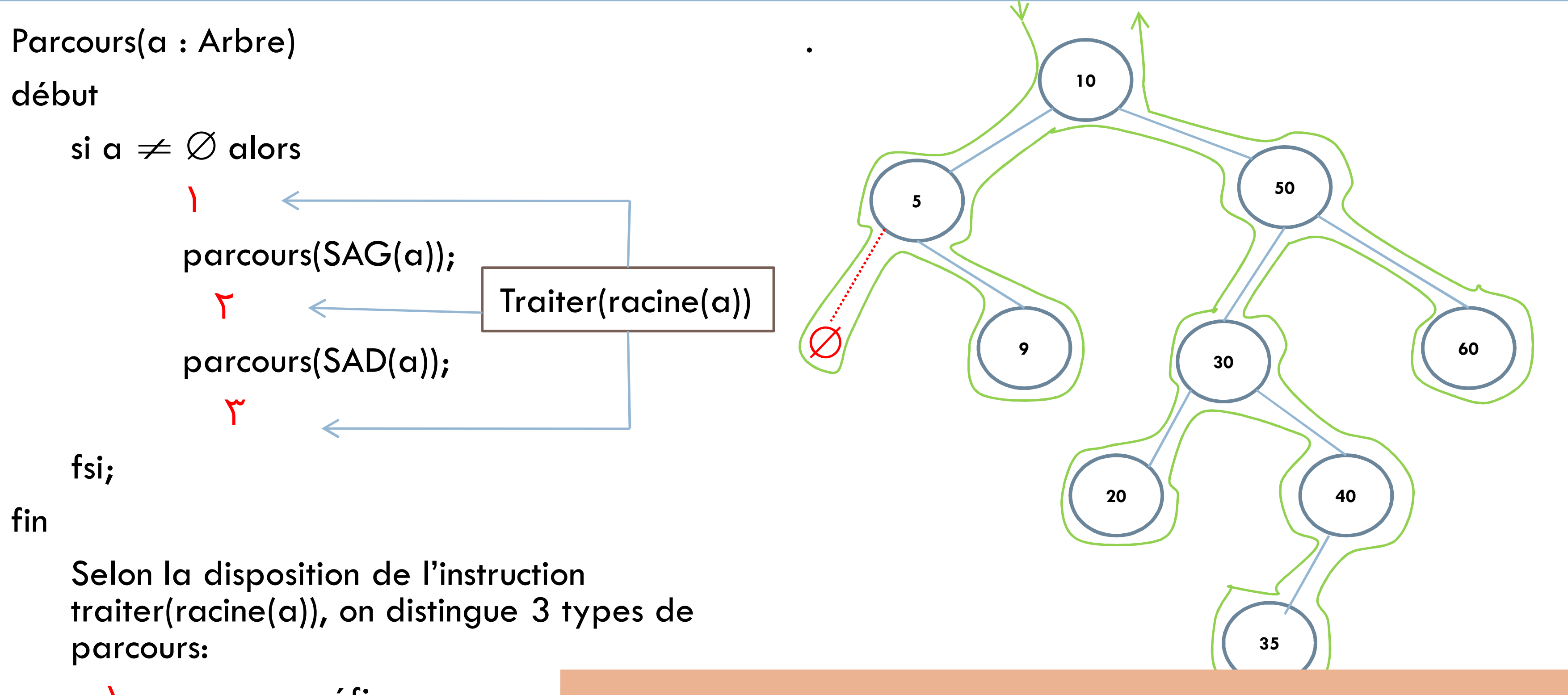
- la hauteur est, en moyenne, un O(log n) et un O(n) dans le pire des cas.

- Les algorithmes sur les arbres binaires se ramènent souvent aux algorithmes de parcours de ces arbres.

Algorithmes de parcours

puis on ferme la parenthèse

```
Parcours(a : Arbre)
début
    si a ≠ ∅ alors
            ١
            parcours(SAG(a));
              ٢
            parcours(SAD(a));
              ٣
    fsi;
fin
```



Selon la disposition de l'instruction traiter(racine(a)), on distingue 3 types de parcours:

- ١ : parcours préfixe.
- ٢ : parcours infixe.
- ٣ : parcours postfixe.

On passe 3 fois sur un nœud .

Parcours préfixe: on traite un nœud lorsqu'on le rencontre pour la 1er fois.

10-5-9-50-30-20-40-35-

Parcours infixe: " " " " la 2è fois.

5-9-10-20-30-35-40-50-60

Parcours posfixe: " " " le quitte pour la dernière fois.

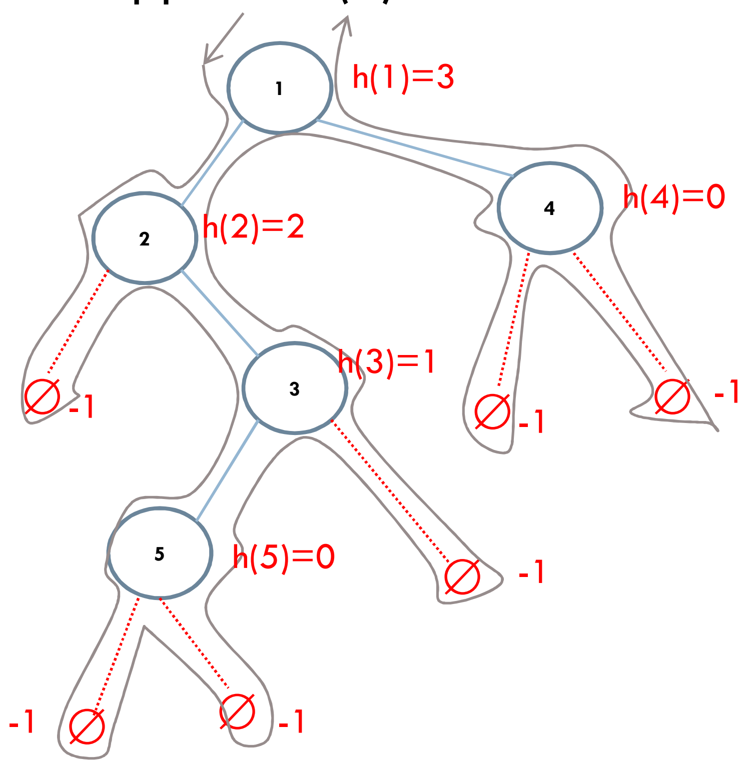
9-5-20-35-40-30-60-50-

Récursivité double
&
Arbre des appels récursifs

- Exemple1: calcul de la hauteur d'un arbre binaire.

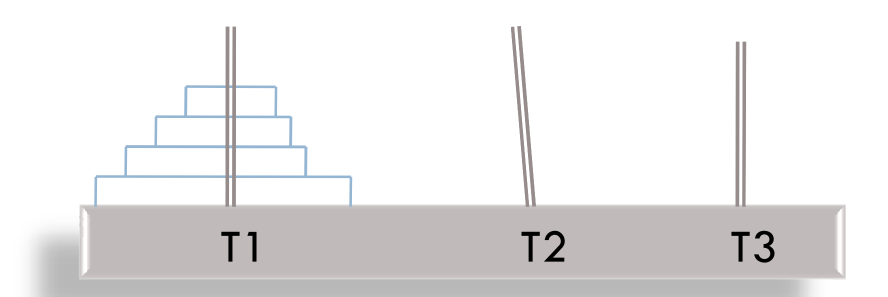
```
    h(a : arbre)
début
    si a = ∅ alors retourner -1
    sinon
            h1 := h(SAG(a));
            h2 : h(SAD(a));
            retourner(1+max(h1,h2));
    fsi;
fin
```

- Un arbre est donné par sa racine
- appel de h(1):



- Exemple2 : **tours de Hanoï** (**Occupation des moines de Hanoï**)

  Le jeu consiste à faire passer les disques de la tour T1 à la tours T2, en ne déplaçant qu'un seul disque à la fois, et en utilisant la tour intermédiaire T3 de telle sorte qu'à aucun moment un disque ne soit empilé sur un disque de plus petite dimension.



La solution semble difficile, et pourtant une solution récursive existe.

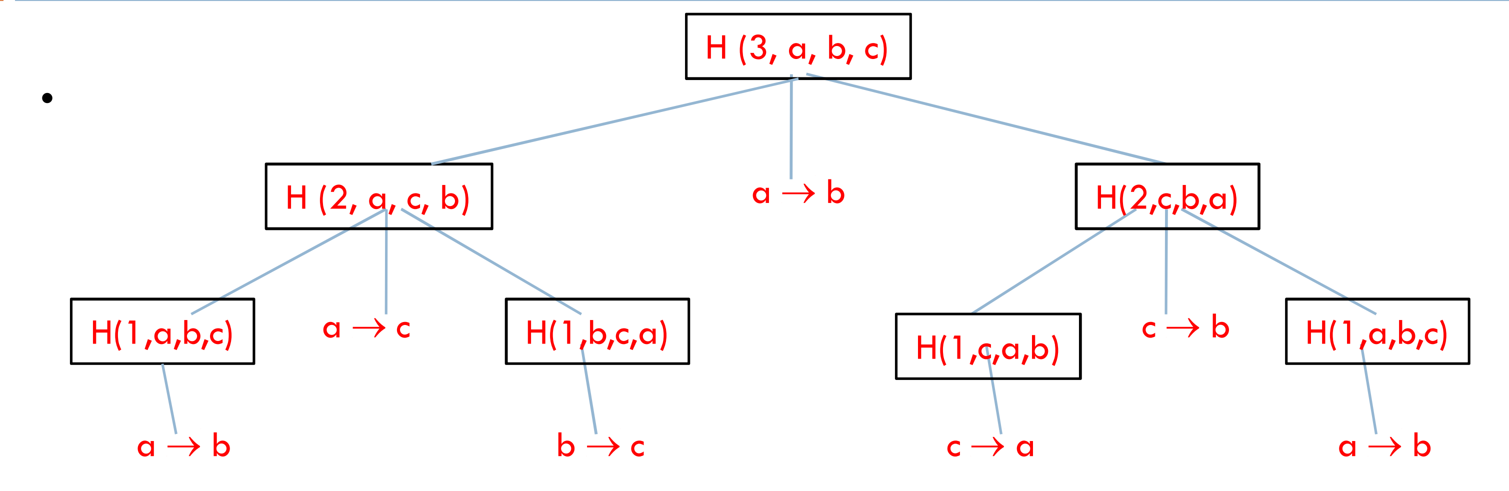
Soit n le nombre de disques à déplacer. Si n=1 la solution est triviale.
Si on sait transférer n-1 disques alors on sait en transférer n.
Il suffit de transférer les n-1 disques supérieurs de la tours T1 vers la tours T3, de déplacer le disque le plus grand de T1 vers T2, puis de transférer les n-1 disques de T3 vers T2. Ceci se traduit par l'algorithme récursif suivant:

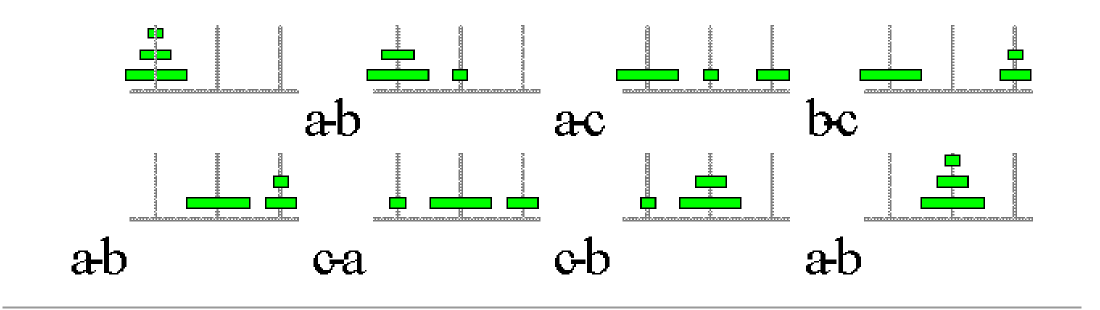
Tours de Hanoï

```
H(n,T1,T2,T3)
début
   si n = 1 alors écrire(T1, '→',T2)
   sinon
         H(n-1,T1,T3,T2);
         écrire(T1,'→',T2)
         H(n-1,T3,T2,T1);
   fsi
fin
```

```
H(n,T1,T2,T3)
début
  si n = 1 alors écrire(T1, '→',T2)
  sinon
      H(n-1,T1,T3,T2);
      écrire(T1,'→',T2)
      H(n-1,T3,T2,T1);
  fsi
fin
```

Arbre des appels de H(3,a,b,c)





Tours de Hanoï : Complexité

Soit T(n) le temps pour déplacer les n disques. T(n) vérifie l'équation :

$$\begin{cases} T(1) = 1 \\ T(n) = 2\,T(n-1) + 1 \end{cases}$$

On a :

$T(2) = 2 + 1$

$T(3) = 2(2 + 1) + 1 = 2^2 + 2 + 1$

On montre, par récurrence, que

$T(n) = 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$

- Sachant que $T(10) = 2^{10} - 1 = 1023$ et une année $\cong 0.3 \times 10^8$ secondes, il faudrait, pour les moines, 10^{10} siècles pour pouvoir déplacer 64 disques!

Méthode « Diviser pour Résoudre »

E. CHABBAR

DVR

- Les algorithmes sont regroupés en familles selon certains concepts t. q. Division pour Résoudre, Glouton, Programmation dynamique.
- \- L'aspect DVR consiste à diviser le problème initial (de taille n) en sous-problèmes similaires de tailles plus petites (en général de taille n/2, n/3,…).

  \- Ces sous-problèmes sont résolus récursivement.

  \- On peut combiner ces solutions récursives pour avoir la solution du problème initial.

DVR
Exemples

1. Calcul de x^n ($n \geq 1$)

- Le nombre de multiplications par la méthode classique ($x^n = x\, x^{n-1}$) est de l'ordre de n.
- La méthode DVR exploite la définition suivante:

$$x^n = \begin{cases} 1 & \text{si } n=1 \\ x^p \,.\, x^p & \text{si } n=2p \\ x \,.\, x^p \,.\, x^p & \text{si } n=2p+1 \end{cases}$$

- pour calculer x^n on fait appel (récursif) au calcul x^p ou $p = E(n/2)$. (algo. Slide suivant)

- si T(n) est le nombre de multiplications pour calculer x^n alors $T(n) = T(n/2) + 1$ si n est pair ou $T(n) = T(n/2) + 2$ si n est impair. On a, dans tous les cas, $T(n) = T(n/2) + O(1)$.

Comme il y a $\log_2 n$ divisions euclidiennes successives (avant d'avoir 1 comme quotient) , donc **$T(n) = O(\log(n))$**. (pour s'en convaincre prendre n une puissance de 2)

$(T(n)=O(1)+O(1)+\ldots+O(1)$ $\log_2 n$ fois)

DVR
Exemples

1. **Calcul de x^n ($n \geq 1$)**

```
 puissBin(x,n)
Début
        si n = 1 alors retourner x
        sinon
              si n mod 2 = 0 alors
                            y := puissBin(x,n/2)
                           retourner (y*y)
              sinon
                            y := puissBin(x,n/2)
                           retourner (x*y*y)
              fsi;
        fsi;
Fin.
```

DVR
Exemples

2. RECHERCHE DICHOTOMIQUE

- La recherche séquentielle d'un élément x dans un tableau A à n éléments est de l'ordre de n.
- La recherche dichotomique exige que le tableau soit **trié**. Elle consiste à:

  - comparer x à l'élément du milieu

  - si c'est différent, x peut se trouver soit dans la moitié gauche soit dans la moitié droite du tableau A, selon que x < A[milieu] ou x> A[milieu].

2. **Recherche dichotomique**

```
 rechDicho(A,inf,sup,x)
début
        si inf ≤ sup alors
              m := (inf + sup)/2
              si x = A[m] retourner m;
                            si x < A[m] alors retourner rechDicho(A,inf,m-1,x);
                            sinon  retourner rechDicho(A,m+1,sup,x);
                            fsi;
              fsi;
        sinon retourner 0;
        fsi;
fin
```

DVR
Exemples

2. **Recherche dichotomique**

- Complexité de la recherche dichotomique

Dans le pire des cas (i.e. x n'est pas dans le tableau) le nombre de comparaisons T(n), pour chercher x dans un tableau A[1..n] à n éléments, vérifie:

$$T(1) = 1$$

$$T(n) = T(n/2) + 1 \quad , \quad n>1$$

- la solution de cette équation dépend du nombre d'itérations (nombre de divisions par 2)

DVR
Exemples

Itération	Nbre d'élts du ss tableau
0	n
1	n/2
2	n/4
3	n/8
.	.
.	.
p	$n/2^p$

L'algorithme utilise p itérations(et aussi p comparaisons) et s'arrête lorsque $n/2^p = 1$, c.à d. $p = \log_2 n$.

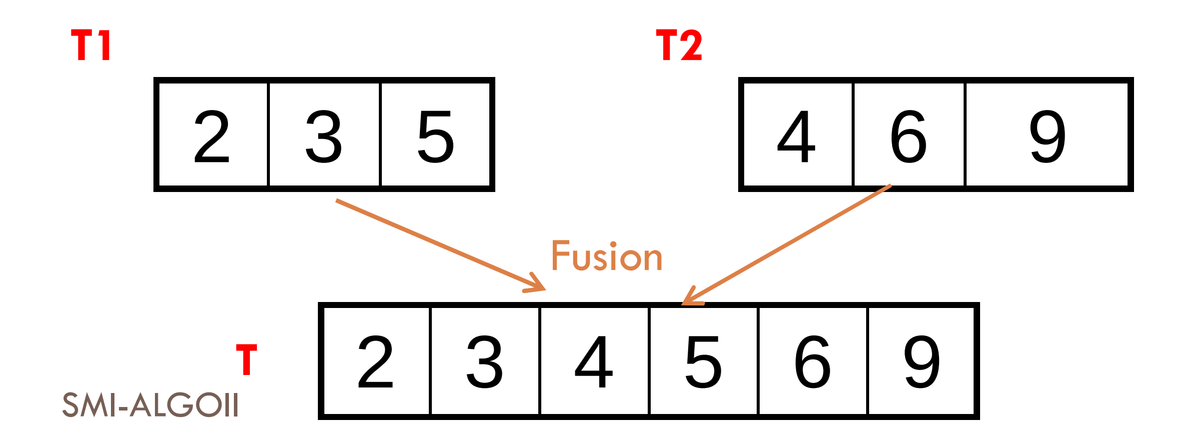
par conséquent, la recherche dichotomique est en $O(\log_2 n)$, i.e.

$$T(n) = O(\log_2 n)$$

3. Tri par fusion

- Fusion de deux tableaux triés.

Etant donnés deux tableaux triés T1[1..n1] et T2[1..n2]. La fusion consiste à construire un tableau T[1..n1+n2] contenant tous les éléments de T1 et T2 dans l'ordre croissant.



Fusion


```
Fusion(T1,n1,T2,n2)
// T est un tableau qui contient le résultat de la fusion
 i1:=1; i2:= 1; k:=1;
 tantque (i1≤n1) et (i2≤n2) faire
     si T1[i1] ≤ T2[i2] alors
                T[k] := T1[i1];
                 k:=k+1;  i1 := i1 +1;
     sinon
                T[k] := T2[i2];
                k:=k+1;  i2 := i2 + 1;
     fsi;
 ftantque;
 // on recopie les élts restants dans l'un des //tableaux Ti dans le tableau T
 si  i1 ≤ n1  alors  copier(T1,i1,n1,T,k)
 sinon          copier(T2,i2,n2,T,k)
 fsi;
retourner (T);
```

```
copier(A,d,f,B,j)
début
          pour i:=d à f  faire
                         B[j] := A[i]
                         j := j+1;
          fpour;
retourner(B);
fin
```

la complexité de la fusion est en O(n1+n2)

TriFusion

- Le tri par fusion est un exemple typique de la stratégie diviser pour résoudre, à savoir:

1. Diviser le tableau T[1..n] en deux sous-tableaux T[1..E(n/2)] et T[E(n/2)+1..n]
2. Trier (récursivement) les deux sous-tableaux (2 appels récursifs à la même fonction TriFusion).
3. Fusionner les deux sous-tableaux;

Ceci est schématisé par l'exemple suivant:

TriFusion

-

Diviser

5	3	2	9	4	6

5	3	2

9	4	6

Résoudre récursivement

Tri récursif

2	3	5

4	6	9

Combiner

2	3	4	5	6	9

TriFusion


```
TriFusion(T, n)
// T1, T2 : des tableaux (locaux) de longueurs variables à chaque appel
début
    si n > 1 alors
            copier(T, 1, n/2, T1,1);
            copier(T, n/2 +1, n, T2, 1);
            T1 := TriFusion(T1, n/2);
            T2 := TriFusion(T2, n – n/2);
            T := Fusion(T1, n/2, T2, n – n/2);
    fsi;
retourner(T);
fin
```

Retourner(Fusion(TriFusion(T1,n/2),n/2,TriFusion(T2,n-n/2),n-n/2)

TriFusion: exemple d'exécution

Tableau à trier:

3	9	1	6	5	2	4	8

Recopie de ss-tableau

Tableau (vert)retourné à l'appel récursif

Fusion des 2 ss-tableaux(rouges) résultats des appels récursifs

DVR: Compléxité

- La complexité T(n) pour trier un tableau de n éléments par l'algorithme TriFusion vérifie:

$$\begin{cases} T(1) = 0 \\ T(n) = 2\,T(n/2) + O(n) \;,\; n > 1 \end{cases}$$

(Il y a 2 appels récursifs, chacun porte sur la moitié du tableau. O(n) pour recopier les 2 ss-tableaux en 2xO(n/2) + leur fusion en O(n))).

Equation de récurrence des alg.DVR

- La récurrence, utilisée par les algorithmes type DVR, est souvent de la forme:

$$T(n) = \begin{cases} O(1) & \text{pour } n=1 \\ a\,T(n/b) + O(n^d) & , n > 1 \end{cases}$$

a : est le nombre de divisions du problème initial en sous-problème (nombre d'appels récursifs)

n/b : la taille de chaque sous-problème ($b \geq 2$)

$O(n^d)$: le temps nécessaire pour décomposer le problème en sous-problème + le temps pour combiner les solutions des ss-problèmes pour avoir la solution du problème initial.

DVR: Résultat de Complexité

- **Théorème:**

Soit T : IN IR+ une fonction croissante à partir d'un certain rang, telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $d \geq 0$, $a > 0$ pour lesquels

$$T(n_0) - k$$

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d \qquad n > n_0, \quad \frac{n}{n_0} \text{ puissance de } b$$

Alors on a :

$$T(n) = \begin{cases} O(n^d) & \text{si } a < b^d \\ O(n^d \log_b n) & \text{si } a = b^d \\ O(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

DVR: Résultat de Complexité

soit $T(n) = aT\left(\frac{n}{b}\right) + cn^d$.

On montre par récurrence sur p (p≥1), que : $T(n) = a^p\, T\left(\frac{n}{b^p}\right) + cn^d \sum_{j=0}^{p-1} \left(\frac{a}{b^d}\right)^j$

prenons $\frac{n}{n_0} = b^p$*, et donc* $p = O(\log_b n)$. *La relation prédente devient :*

$$T(n) = ka^p + cn^d \sum_{j=0}^{p-1} \left(\frac{a}{b^d}\right)^j = f(n) + cn^d g(n)\text{, où :}$$

$$f(n) = ka^p = O\left(n^{\log_b a}\right) \quad et \quad g(n) = \sum_{j=0}^{p-1} \left(\frac{a}{b^d}\right)^j$$

du fait que : $a^p = e^{p\ln a} = e^{\log_b \frac{n}{n_0} \ln a} = e^{\ln\frac{n}{n_0} \frac{\ln a}{\ln b}} = \left(\frac{n}{n_0}\right)^{\log_b a} = O(n^{\log_b a})$

$g(n)$ *est une suite géométrique de raison* $r = \frac{a}{b^d}$, *par suite* $g(n) = \frac{r^p - 1}{r - 1}$ $(r \neq 1)$

premier cas: si $r = \frac{a}{b^d} = 1$, *alors* $g(n) = p = O(\log_b n)$ *et par conséquent* $T(n) = O\left(n^d \log_b n\right)$ *si* $a = b^d$

deuxième cas: $g(n) = \frac{1 - r^p}{1 - r} \leq \frac{1}{1 - r}$ *si* $r = \frac{a}{b^d} < 1$ *et* $g(n) = O(1)$ *dans ce cas ,* *par conséquent* $T(n) = O(n^d)$ *si* $a < b^d$

troisième cas: $g(n) = \frac{r^p - 1}{r - 1} \leq r^p$ *si* $r = \frac{a}{b^d} > 1$ *et* $g(n) = O\left(n^{-d}\, n^{\log_b a}\right)$ *du faite que:*

$$r^p = \frac{a^p}{b^{pd}} = O\left(n^{\log_b a}\right)\left(\frac{n}{n_0}\right)^{-d} = O\left(n^{\log_b a}\right) O(n^{-d})$$

d'où $T(n) = O\left(n^{\log_b a}\right)$ *si* $a > b^d$

DVR: Résultat de Complexité

D'une manière générale. Si

$$\begin{cases} T(1) = O(1) \\ T(n) = a\,T(\frac{n}{b}) + O(n^d) \end{cases} \quad (n>1,\ a > 0,\ b > 1,\ d \geq 0)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \text{si } a < b^d \\ O(n^d \log_b n) & \text{si } a = b^d \\ O(n^{\log_b a}) & \text{si } a > b^d \end{cases}$$

DVR

- Applications

1. Pour la recherche dichotomique et le calcul de la puissance on a : $T(n) = T(n/2) + O(1)$

   donc $T(n) = O(\log n)$ $(a=1 , b=2, d=0)$

2. Pour le TriFusion on a : $T(n) = 2\,T(n/2) + O(n)$

   Alors $T(n) = O(n \log_2 n)$ $(a=2, b=2, d=1)$

3. $T(1) = 1$

   $T(n) = 2t(n/2) + O(n^2)$

   a pour solution $T(n) = O(n^2)$ $(a=b=d=2)$

DVR : tri rapide (quickSort)

- Soit à trier le tableau T[g..d] (au départ g=1 et d=n)
- Le principe de l'algorithme réside dans une procédure, appelée **partition**, qui réorganise les éléments de T autour d'un pivot (élément du tableau choisi au hasard) de sorte que :

  1) Il existe un indice p ($g \leq p \leq d$) tel que p est la position définitive du pivot (T[p] = pivot).

  2) tous les éléments T[g], …, T[p-1] sont inférieurs ou égaux à T[p].

  3) tous les éléments T[p+1], …T[d] sont supérieurs ou égaux à T[p].

DVR : tri rapide

- Le travail de la fonction partition consiste à:
  - Choisir un élément du tableau comme pivot(par exemple le premier T[g])
  - Parcourir le tableau depuis la gauche(de gauche à droite) jusqu'à rencontrer un élément ≥ T[g]
  - Parcourir le tableau depuis la droite (de droite à gauche) jusqu'à rencontrer un élément ≤ T[g]
  - Echanger ces deux éléments dans le tableau
  - Continuer ce processus jusqu'à ce que les deux indices (de gauche et de droite) se croisent.
  - Finalement, échanger le pivot T[g] et l'élément indiqué par l'indice de droite.

≤ pivot
Non encore analysé
≥ pivot

g → i
j ← d

DVR : Tri Rapide


```
//T[n+1] = + ∝
Partition(T, g, d)
début
    pivot := T[g];
    i := g;  j := d+1;
    tantque  i < j  faire
        i := i +1;
         tantque  T[i] < pivot  faire  i:=i+1;
         ftantque
        j := j -1;
         tantque  T[j] > pivot  faire  j:=j-1;
         ftantque
        si  i < j  alors  échanger(T, i, j); fsi
    ftantque
    échanger(T, g, j);
    retourner(j)
fin
```

- Exemple

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	i	j
3	1	4	1	5	9	2	6	5	3	5	8	9		
3	1	4	1	5	9	2	6	5	3	5	8	9	3	10
3	1	**3**	1	5	9	2	6	5	**4**	5	8	9	3	10
3	1	3	1	5	9	2	6	5	4	5	8	9	5	7
3	1	3	1	**2**	9	**5**	6	5	4	5	8	9	5	7
3	1	3	1	**2**	9	5	6	5	4	5	8	9	6	5
2	1	3	1	**3**	9	5	6	5	4	5	8	9		5
2	1	3	1	3	9	5	6	5	4	5	8	9		

DVR : Tri Rapide

- La fonction partition, appliquée à un tableau T, produit trois sous-tableaux:
  - Un sous-tableau réduit à un seul élément T[p] qui garde sa place définitive dans le tri de T, et
  - Deux sous-tableaux T[g .. p-1], T[p+1 .. d].
- Pour trier T, il suffit d'appliquer récursivement le même algorithme sur les deux sous-tableaux.

DVR : Tri Rapide


```
quickSort(T, inf, sup)
début
   si  inf  < sup  alors
         p := partition(T, inf, sup);
         quickSort(T, inf, p-1);
         quickSort(T, p+1, sup);
   fsi
fin
```

Complexité du Tri rapide

La complexité de la fonction partition, appliquée à T[1..n], est en O(n).

- Cas le plus défavorable :

Cas où le pivot sort, à chaque fois, en premier (ou en dernier) élément (T: tableau trié).

La partition coupe le tableau en un morceau de un élément et un morceau de n-1 éléments, dans ce cas on a :

$$C(n) = C(n-1) + O(n)$$

(O(n) est le coût de la partition)

On en déduit que C(n) est en $O(n^2)$

Complexité du Tri rapide

- **Cas le plus favorable :**

cas où le pivot est l'élément médiane de T.

La partition coupe T en deux morceaux de taille n/2

$C(n) = 2\ C(n/2) + O(n)$

ce qui donne: $C(n) = O(n \log n)$

La complexité moyenne est aussi de l'ordre de n log n

Complexité du Tri rapide

- Complexité moyenne du tri rapide

La formule de récurrence donnant le nombre de comparaisons effectuées par le tri rapide pour une permutation aléatoire de n éléments vérifie :

$C_0 = C_1 = 0$ et

$$C_n = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C_i + C_{n-i-1}) \qquad \text{pour } n \geq 2$$

Le terme générique C_n peut s'écrire :

$$C_n = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C_i$$

Complexité moyenne du tri rapide

- $C_n = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C_i$

$$C_n = n - 1 + \frac{2}{n} C_{n-1} + \frac{2}{n} \sum_{i=0}^{n-2} C_i$$

$$C_n = n - 1 + \frac{2}{n} C_{n-1} + \frac{n-1}{n} \left(n - 2 + \frac{2}{n-1} \sum_{i=0}^{n-2} C_i\right) - \frac{(n-1)(n-2)}{n}$$

$$C_n = \frac{2}{n} C_{n-1} + \frac{n-1}{n} C_{n-1} + \frac{2n-2}{n}$$

$$C_n = \frac{n+1}{n} C_{n-1} + \frac{2(n-1)}{n}$$

En posant $D_n = \frac{C_n}{n+1}$ On aura la récurrence : $D_n = D_{n-1} + \frac{2}{n+1} - \frac{2}{n(n+1)}$

En négligeant le dernier terme de D_n on a: $D_n \simeq \log n$

Du fat que : $1 + \frac{1}{2} + \dots + \frac{1}{n} \simeq Ln(n)$ d'où C_n est en $O(n \log n)$

Preuve de programmes

Notions de Logique

E. CHABBAR

Logique

- La logique a joué un rôle décisif dans le développement de l'informatique, notamment en informatique théorique:

  - Définition d'un modèle théorique des premiers ordinateurs (Machine de Turing)

  - Calcul booléen pour la conception et l'étude des circuits.

  - La récursivité pour définir la calculabilité.

  - la décidabilité et la complexité pour étudier la limite de la machine.

  - La programmation fonctionnelle.

Logique

- Actuellement l'informatique a envahi tous les domaines de la vie. Le problème le plus important qui se pose dans la conception d'une application informatique est de prouver qu'elle est exempte d'erreurs et qu'elle résout le problème pour laquelle elle a été conçue. Pour cela, on définit une tache par une formule de logique et on montre qu'elle est satisfaite par un modèle de cette application. Ce type de preuve est appelé « Vérification Formelle ».

Logique

- **Une logique, par définition, est un ensemble de formules.**
- Une formule est construite, sur un alphabet, suivant certaines règles (Syntaxe).
- La sémantique d'une formule est une valuation (ou interprétation dans un modèle) qui détermine la valeur de vérité de la formule.
- Il y a plusieurs types de logiques:

  - Logique des propositions (d'ordre 0)

  - Logique du premier ordre (les prédicats en font partie)

  - logique du second ordre et logique d'ordre supérieur.

Logique des propositions

- - On note P = {p, q, …} l'ensemble des propositions atomiques. Chaque proposition atomique est une variable qui ne peut prendre que « vrai » ou « faux ».

  - {$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$} l'ensemble des connecteurs (ou opérateurs) logique plus les parenthèses.
- - On note F(P) l'ensemble des formules déduit de P.
- **F(P) est le plus petit ensemble qui contient P et qui est stable par les connecteurs.**

  En d'autres termes:

  Une formule est une suite de symboles de P $\cup$ {$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, (,)} construite selon les règles suivantes:

Logique des propositions

- Toute proposition atomique est une formule.
- si f est une formule alors $\neg f$ est aussi une formule.
- si f1 et f2 sont des formules alors :
  - $f1 \wedge f2$ est une formule
  - $f1 \vee f2$ « « «
  - $f1 \Rightarrow f2$
  - $f1 \Leftrightarrow f2$ « « «

Exemple: $(\neg p \Rightarrow q) \vee (p \wedge q)$ est une formule.

Logique des propositions

- La sémantique des opérateurs logiques est donnée par des tables de vérité.

p	q	$\neg p$	$\neg q$	$p \vee q$	$p \wedge q$	$p \Rightarrow q$ $(\neg p \vee q)$
0	0	1	1	0	0	1
0	1	1	0	1	0	1
1	0	0	1	1	0	0
1	1	0	0	1	1	1

Prédicats

- Les prédicats sont construits avec :
  - les constantes (0,1,2,….)
  - les variables (x, y, ….)
  - les fonctions (f, g, +, *, …)
  - des connecteurs logiques ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$)
  - des parenthèses
  - des quantificateurs ($\forall$, $\exists$)
- Un prédicat atomique est un prédicat qui ne contient ni connecteur ni quantificateur.
  - Exemples: $x < y$; pair(2 x) ;

Prédicats

- Règles de formations des formules de prédicats:

  - Tout prédicat atomique est une formule

  - si f1 et f2 sont des formules alors

    $\neg$ f1, f1 $\vee$ f2, f1 $\wedge$ f2, f1 $\Rightarrow$ f2, f1 $\Leftrightarrow$ f2

    sont des formules.

  - si f est une formule alors

    $\forall$ x f est une formule

    $\exists$ x f est une formule

Prédicats

- **Exemples:**

  1) $\forall x \; pair(x + x)$
  2) $\exists x \; premier(x) \wedge \; x < succ(succ(0))$
  3) $\forall x \; (oiseau(x) \Rightarrow vole(x))$ (tous les oiseaux volent)
  4) $\exists x \; (oiseau(x) \wedge \neg \; vole(x))$ $(4 \equiv \neg \; 3)$
  5) $\forall i \; \forall j \; (1 \leq i \leq j \leq n \Rightarrow T[i] \leq T[j])$ (spécification d'un tableau trié)

- **Remarques**

  $\neg \; (\forall x \; P) \equiv \exists x \; \neg P$

  $\neg \; (\exists x \; P) \equiv \forall x \; \neg P$

Déduction logique

- Démonstrations logiques

- Un séquent est un couple de la forme ($\mathcal{J}$,**f**), où **f** est une formule et $\mathcal{J}$ un ensemble fini de formules . L'ensemble $\mathcal{J}$ est l'ensemble des prémisses du séquent, la formule **f** sa conclusion.

- Séquents prouvables
- Un séquent ($\mathcal{J}$,**f**) est prouvable, ce que l'on notera

$\mathcal{J} \vdash$ **f**, s'il peut être construit en utilisant un nombre fini de fois les 6 règles suivantes :

1. si **f**$\in\mathcal{J}$, alors $\mathcal{J} \vdash$**f** (Hypothèse)
2. si **g** $\notin \mathcal{J}$ et $\mathcal{J} \vdash$ **f**, alors $\mathcal{J}$, **g** $\vdash$ **f**
3. si $\mathcal{J} \vdash$ (**f** $\Rightarrow$ **f'**) et $\mathcal{J} \vdash$ **f**, alors $\mathcal{J} \vdash$ **f'** (Modus Ponens)
4. si $\mathcal{J}$, **f** $\vdash$ **f'**, alors $\mathcal{J} \vdash$(**f** $\Rightarrow$ **f'**) (Synthèse)
5. $\mathcal{J} \vdash$ **f** ssi $\mathcal{J} \vdash$ **¬¬f**
6. si $\mathcal{J}$,**f** $\vdash$ **f'** $\mathcal{J}$,**f** $\vdash$ **¬f'** alors $\mathcal{J} \vdash$**¬f** (Raisonnement par l'absurc (Contradiction)

- Démonstrations logiques
- Une démonstration d'un séquent prouvable $\mathcal{J} \vdash \mathbf{f}$ est une suite finie de séquents prouvables $\mathcal{J}_i \vdash \mathbf{f}_i$,

  i = 1,...,n, telle que :
- $\mathcal{J}_n = \mathcal{J}$ et $\mathbf{f}_n = \mathbf{f}$
- chaque séquent de la suite est obtenu à partir des séquents qui le précède en appliquant l'une des 6 règle
- Remarque : le premier séquent de la suite est nécessairement obtenu par utilisation d'une hypothèse

- Démonstrations logiques - Exemple
- Tous les hommes sont mortels, et
- les Grecs sont des hommes, donc les Grecs sont mortels

h = « être un homme »

m = « être mortel »

g = « être Grec »

$(h \Rightarrow m), (g \Rightarrow h) \vdash (g \Rightarrow m)$

- Démonstrations Logiques - Exemple
- Tous les hommes sont mortels, et
- les Grecs sont des hommes, donc les Grecs sont mortels
- 1. $(h \Rightarrow m), (g \Rightarrow h), g \vdash g$ hypothès
- 2. $(h \Rightarrow m), (g \Rightarrow h), g \vdash (g \Rightarrow h)$ hypothès
- 3. $(h \Rightarrow m), (g \Rightarrow h), g \vdash h$ MP (1&2)
- 4. $(h \Rightarrow m), (g \Rightarrow h), g \vdash (h \Rightarrow m)$ hypothès
- 5. $(h \Rightarrow m), (g \Rightarrow h), g \vdash m$ MP (3&4)
- 6. $(h \Rightarrow m), (g \Rightarrow h) \vdash (g \Rightarrow m)$ synthèse

Preuve de programmes

Logique de Hoare

Correction de programmes / Spécification

- un programme est correct s'il effectue la tâche qui lui est confiée dans tous les cas permis possibles

- nécessité de disposer d'un langage de spécification permettant de décrire **formellement** la tâche confiée à un programme.

Correction de programmes / Spécification

- description des propriétés que doit satisfaire un programme pour répondre au problème posé

  - relation entre les entrées et les sorties du programme

Entrée E → Programme → Sortie S

Précondition : P(E)

Postcondition : Q(E, S)

Correction de programmes / Spécification

- Exemple

une spécification pour le problème du calcul de la racine carrée entière par défaut :

Données n : **entier ;**
Résultat r : **entier ;**

Pré condition : $n \geq 0$

Algorithme: r := 0; tantque ($n \geq (r+1)^2$) faire r := r + 1 ftantque

Post condition : $(r^2 \leq n) \wedge (n < (r+1)^2)$

Test vs Vérification

- Les couples (n, r) de l'ensemble suivant {(0, 0), (1, 1), (2,), (3, 1), (4, 2), …} sont des tests qui réussissent

- Hoare propose une exécution de l'algorithme sur une valeur symbolique qui est un ensemble de valeurs défini par une expression logique (ou prédicat).

- L'ensemble {(0, 0), (1, 1), (2, 1), (3, 1), (4, 2), …} est défini par {(n,r) / $\mathbf{n \geq 0 \wedge r \geq 0 \wedge r^2 \leq n \wedge (n < (r+1)^2}$}

- On note $\mathbf{\{n \geq 0 \wedge r \geq 0 \wedge r^2 \leq n \wedge (n < (r+1)^2\}}$ la valeur symbolique des variables n et r.

- L'exécution d'un algorithme A sur une valeur symbolique de données définies par l'expression { p} qui donne une valeur symbolique résultat définie par l'expression { q} est notée {p} A {q} . **{p} A {q} est appelé triplet de Hoare (p : précondition, q : postcondition).**

Système formel

- **Interpretation d'un triplet de Hoare**

  {p} A {q} signifie :
  Si la propriété p est vraie avant l'exécution de A ET si **l'exécution** de A **se termine**, ALORS la propriété q est vraie après l'exécution de A.
  (Correction partielle)

- Un **système formel** est un triplet <L, Ax, R> où :
  - L est un langage définissant un ensemble de formules,
  - Ax est un sous-ensemble de L ; chaque formule de Ax est appelée axiome,

R est un ensemble de règles de déduction de formules à partir d'autres formules

Langage algorithmique

- Instructions :

  - **Affectation** (:= symbole d'affectation et = pour la comparaison)
  - **Contrôle** :

    Si <cond> **alors** instruction **fsi**

    ou **Si** <cond> **alors** instruction
    sinon instruction **fsi**
  - **Boucle** :

    Tantque <cond> **faire** instruction **ftantque**

- Restriction :

  - Pas de fonction (ou procédure)
  - ni de variable pointeur
  - pas de désignation de la forme t[i] où i fait référence à un autre tableau.

Logique de Hoare

- Définition de la logique de Hoare : La logique de HOARE est un triplet <L, Ax, R> avec :

  - L est l'ensemble des formules { p} A { q} où p et q sont des prédicats et A est un fragment d'algorithme(ou de programme)

  - Ax est l'ensemble des axiomes de la forme suivante : { p[x / e] } x := e { p(x)}

  (p(x) est obtenu de p[x/e] en substituant toute occurrence de e par x)

  - R est l'ensemble de règles de déduction suivant :

Logique de hoare

- (Séq) si {p} A1 {r} ; {r} A2 {q} alors
  {p} A1 ; A2 {q}
- ($cons_g$) si $p \Rightarrow p'$, {p'} A {q} alors
  {p} A {q}
- ($cons_d$) si {p} A {p'}, $p' \Rightarrow q$ alors
  {p} A {q}
- ($cond_1$) si $\{p \wedge c\}$ A {q} , $(p \wedge \neg c) \Rightarrow q$ alors
  {p} si c alors A fsi {q}
- ($cond_2$) si $\{p \wedge c\}$ A1 {q} , $\{p \wedge \neg c\}$ A2 {q} alors
  {p} si c alors A1 sinon A2 fsi {q}
- (tantque) si $\{I \wedge C\}$ A {I} alors
  { I} tantque C faire A ftantque $\{I \wedge \neg C\}$

(I doit être un invariant de la boucle)

Exemple : règles d'affectation et conséquence

•

- $\{ p[x / e] \} \quad x := e \quad \{ p(x)\}$

1. $\{n+1> 0\}$ $n:= n+1$ $\{n>0\}$ (x=n, e = n+1)
2. $\{k > 0\}$ $n :=0$ $\{k > n\}$ (x=n, e =0)
3. $\{x = 4\}$ $x := x+1$ $\{x=5\}$ en écrivant:
   $\{(x+1) -1 =4\}$ $x := x+1$ $\{ x - 1 =4\} \Rightarrow \{x = 5\}$

4. $\{x \geq 0\}$ $x := x + 1$ $\{x \geq 1\}$

   $\{x \geq 0\} \Rightarrow \{x + 1 \geq 1\}$
   $x := x + 1$
   $\{x \geq 1\}$

Exemple : règles de condition

5. {vrai}

si $x \geq 0$ alors

$\{vrai \wedge x \geq 0\} \Rightarrow \{x \geq 0\} \Rightarrow \{x+1 \geq 1\}$

x := x + 1;

$\{x \geq 1\}$

sinon

$\{vrai \wedge x < 0\} \Rightarrow \{x < 0\} \Rightarrow \{-x > 0\}$

x := - x;

$\{x > 0\} \Rightarrow \{x \geq 1\}$

fsi

$\{x \geq 1\}$

Schéma de preuve d'un algorithme itératif Tant que

- Soit l'algorithme suivant:

```
début
  {pré}
  init;
  {I}
  tantque C faire
        {I ∧ C}
        A
        {I}
  ftantque
  {I ∧ ¬ C} ⇒
  {post}
```

Exemple: calcul de la racine carrée par défaut

- Algorithme A:

donnée : n : entier;

résultat : r : entier;

début

r := 0;

tantque $n \geq (r + 1)^2$ **faire**

r := r + 1;

ftantque

fin

- **Spécification:**

précondition : $\{n \geq 0\}$

postcondition : $\{ (r^2 \leq n \wedge n < (r + 1)^2) \}$

- **invariant de la boucle:** $I = \{r^2 \leq n\}$ (évident, sinon on le déduit de la postcondition)

Algorithme A annoté et prouvé

Algorithme A:

donnée : n : entier;

résultat : r : entier;

Début

$\{n \geq 0\} \Rightarrow \{n \geq 0*0\}$

r := 0;

$\{n \geq r*r\}$

tantque $n \geq (r + 1)^2$ **faire**

$\{n \geq r^2 \wedge n \geq (r + 1)^2\}$

r := r + 1;

$\{n \geq r^2\}$

ftantque

$\{r^2 \leq n \wedge n < (r + 1)^2\}$

fin

Exemple: calcul de n!

- **Algorithme B:**

```
 donnée :  n : entier
 résultat :  y : entier
début
   x := n;
   y := 1;
   tantque   x > 1  faire
            y := y * x;
            x := x – 1;
   ftantque
fin
```

- **Spécification:**
  précondition : $\{n \geq 0\}$
  postcondition : $\{y = n!\}$
- **Invariant:** $I = \{ n! = y\ x! \ \wedge x \geq 0\}$

Algorithme B annoté et prouvé

Algorithme B:

donnée: n : entier;

résultat: y : entier;

début

$\{n \geq 0\}$

x := n;

$\{x \geq 0 \wedge x = n\}$

y := 1;

$\{n! = y\ x! \wedge x \geq 0\}$

tantque x > 1 **faire**

$\{n! = y\ x! \wedge x \geq 0 \wedge x > 1\} \Rightarrow \{n! = (y\ x)(x-1)! \wedge x > 1\}$

y := y * x;

$\{n! = y\ (x-1)! \wedge x-1 > 0\}$

x := x – 1;

$\{n! = y\ x! \wedge x > 0\} \Rightarrow \{n! = y\ x! \wedge x \geq 0\}$

ftantque

$\{n! = y\ x! \wedge x \geq 0 \wedge x \leq 1\} \Rightarrow \{ y = n!\}$

Exercice: Faites la preuve de l'algo. de n! en faisant une boucle qui va de 1 à n